Python 快速入门精讲

邹琪鲜 编著



扫一扫
免费观看视频
获取全部源代码

ZERO TO HERO

清华大学出版社

Python
快速入门精讲

邹琪鲜 编著

清华大学出版社
北 京

内 容 简 介

本书从零基础入门着手，通过合理的编排，首先引导读者循序渐进地学习 Python 基本语法和语义，再掌握诸如文件和数据库的处理、面向对象编程、开发图形用户界面、网络和多线程编程等实用技术，最后拓展了 Python 的一些热门应用，如大数据和机器学习。

本书通过丰富的案例和真切的经验分享、详细的文字讲解和生动的在线视频演示，给读者带来别样的学习体验。

本书非常适合没有 Python 编程经验的程序员，也适合工作中需要用到程序解决问题的非专业人员，由于独特的编排和讲解，同样适合初学 Python 的学生，甚至可以当作全国计算机二级 Python 的教材使用。总之，对于第一次接触编程的人来说，这是一本非常适合的书。

图书在版编目（CIP）数据

Python 快速入门精讲/邹琪鲜编著. —北京：清华大学出版社，2019
ISBN 978-7-302-51478-7

Ⅰ. ①P…　Ⅱ. ①邹…　Ⅲ. ①软件工具－程序设计　Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 254885 号

责任编辑：王剑乔
封面设计：刘　键
责任校对：李　梅
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：17.25　　**字　　数**：417 千字
版　　次：2019 年 3 月第 1 版　　**印　　次**：2019 年 3 月第 1 次印刷
定　　价：00.00 元

产品编号：077434-01

前 言

在这本书的创作过程中，有过很多的构思，是精炼直接还是面面俱到，是道理连篇还是实用为主……经过深思熟虑，最终的呈现是：涉及技术的地方，用简练的语言去介绍，希望读者能够用最短的时间了解一个新功能或者一个陌生的领域；涉及思想认知方面，用生活化的语言进行详细说明。

读者还会发现本书在介绍一些基础知识的时候，会有一些“废话”。对于有编程经验的人来讲，确实是废话；而对于一个新手来讲，我希望这些“废话”是一个过来人的经验分享，就好像有个老师在你面前娓娓道来，而不是几句话就涵盖很多的概念和知识点，让新手可能想破头也想不通这里面的来龙去脉。我希望通过这样的方式能够帮助新手树立正确的编程观念，继而产生潜移默化的影响。

至于这本书，我希望它是一个指引，而不是什么宝典大全。当读过之后，我希望读者能脱离这本书，形成自己的思维方式和解决问题的方法，而不是总来翻这本书寻找答案。书中不单讲解语言本身，更是注重培养读者的编程意识，通过经验的分享和抛出问题的方式培养读者独立探索和解决新问题的能力，这也是一个合格的程序员所需要具备的素质。

适合读者

零基础的初学者。

掌握其他语言的程序员。

需要编程解决问题的运维工程师。

工作中需要用到程序解决问题的非计算机领域工作者。

编程爱好者。

本科及大专在校学生。

想要报考全国计算机等级考试的读者。

如何学习本书

本书共3篇18章，分别为基础入门篇、进阶应用篇和拓展案例篇。

基础入门篇为第0章至第7章，内容从克服编程恐惧开始，逐步渗透编程思想，建立编程的信心。知识从Python安装开始，以合理的递进式知识结

构编排，介绍了 Python 的数据类型、语句语法以及函数和模块化编程，每个章节的重要知识点都配备了生动的教学视频，可以采取文字和视频同步学习的方式，能更快、更好地吸收新知识。

进阶应用篇为第 8 章至第 15 章，主要介绍文件和数据持久化、面向对象编程、异常处理、开发图形用户界面、正则表达式以及爬虫入门、多线程编程和网络应用编程等相关实用技能，在其中还适当穿插了一些自学任务。通过这部分学习，可以让读者对 Python 的应用有更多了解，增强自学能力，达到学一通三的效果，从而可以在自己的实际工作中快速学习 Python 相关的、新的模块和功能并加以应用。各章节均配备了视频教程。

拓展案例篇是第 16 章至第 18 章，每章通过一个案例讲解某一个领域的应用。因为 Python 的特殊性，Python 在很多领域都有所应用，作为新手可能没办法在短时间内精通 Python 在所有领域的应用。这里选取了三个案例，分别是大数据、语音识别和机器学习入门，每个案例都可以帮助新手对相关领域快速了解并入门，如果对哪个方面感兴趣，或者在工作中有需要，可以再深入进行学习研究，这几个案例都是非常好的入门指南。

另外，在本书中还有一些特别的设计，比如，在基础入门和进阶应用部分有一个叫《英雄无敌》的游戏项目，这其实并不是教你开发游戏的教程，只是通过一种载体让那些看起来无关的语法或者功能有一个有机的结合，对零散知识点与程序设计有一个系统的认知。随着学习的深入，你可以让游戏的功能越来越完善，甚至超出书中的设计，这个游戏可以从最简单的文字形式发展成游戏界面；从单纯的文字到具备复杂故事情节；可以从单机到网络。总之，只要你学了新的知识就可以想想这个功能可以在游戏中用来干什么。在书里很多地方我都做了引导，如果你有想法，就去实现吧！

在本书的"动动手"部分也留了一些开放式的问题，需要自己去考虑和设计，有些做了视频演示，如果想不通，也可以到 QQ 交流群中跟大家进行探讨。

至于实验环境，书中主要的案例都是基于 Python 3＋Windows 10 的环境，个别拓展内容会涉及 Linux 和 Python 2，不过请放心，一定会在你的理解范围内。

相关资源

书中提供了丰富的代码演示，所有的代码都可以在 Github 得到，Github 网址是 https://github.com/milozou/CrazyPython/。代码按章分 18 个目录保存。

如果想跟更多人在线交流经验，可以加入疯狂的 Python QQ 交流群-1：814674076。

如果有内容勘误方面的意见，欢迎直接发邮件至 zouqixian@gmail.com。

最后，希望大家在提到本书时，不会说这是一本 Python 编程的宝典，我希望大家会说这是一本武功心法。看完之后，醍醐灌顶，打通了任督二脉，内功深厚，哈哈。

邹琪鲜

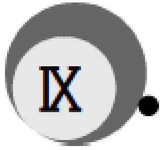
2019 年 1 月



本书配套源代码

目　录 <<

基础入门篇



基础入门篇

第 0 章 从零开始

万事开头难，学习编程似乎也不容易。对于新手来讲，在眼花缭乱的资讯、教程、分享以及五花八门的资源中，首先需要的就是选对学习路线。有太多从入门到放弃的例子，就是因为没有找到学习的主线，把时间精力都浪费在无用的知识和信息上，最后看起来好像学了很多东西，但是真正有用的也就是十分之一。



疯狂的 Python

学习 Python 编程不同于其他计算机技术，并不是罗列出一堆知识点，然后逐一记住它们，也不是同样功能的几个方法都要掌握。很多文章或书籍都会堆砌知识点，相同功能也要列出几个，这就好像孔乙己在说“回”字有几种写法一样。新手初看，感觉内容很丰富，学着学着就没动力了，真的是体验了“学海无涯”。而高手往往会觉得拼凑、罗列知识点的方式过于累赘。其实，学习编程把最主要的核心学会，再掌握解决问题的正确思路，就已经算是入门了，Python 更是如此。

这是个崇尚“最好的办法只有一个”的语言，无须学那么多，遇到需要学习的新功能，运用已经掌握的方法，很快就可以上手并运用。这也是一个合格程序员应该掌握的能力。

0.1 克服编程恐惧

新手学习编程之前通常会有两种情绪：好奇和恐惧。通常小孩子或者纯小白对编程的好奇心会比较大，好奇是因为觉得编程很神奇，可以让计算机帮助我们完成更多的工作；而顾虑比较多的人就会产生恐惧或者畏难心理，恐惧是因为对未知的迷茫和无序。比如很多人在开始学习之前会先说“我的英语不好”“我的数学不好”等类似的问题。

Python 可以说是最适合编程入门的语言了，少儿接触编程，Python 也是最好的选择。既然如此，那就说明这门语言本身其实是非常简单的。

其实，对于新手来说，首先要搞明白的事情就是到底什么是编程，以及明确编程的困难之处到底在哪里。

学编程就好像学习写文章，不同的是，文章是写给人看的，程序是写给计算机看的。我们都有过学习写文章的经历，那是一个过程：从陌生到熟悉的过程，从笨拙到巧妙的过程。没有人是刚认识字就可以写出漂亮文章的。

想想我们学习写作文的过程，最开始的时候并不是一上来老师就跟你说：“来，我们现在学习写作文。”也不是先学汉字或者拼音，我们首先学会的是把想法用语言表达出来，也

就是说，我们最先会的是在脑子里形成想法，那可能是下意识的，也可能是经过深思熟虑产生的想法。有了想法之后，我们可以通过口述表达出来，然后才逐渐开始学习拼音、汉字、词、句子、短语、段落、文章，造句写文章的规矩和语法以及不同类型文章的结构等。

学好中文就可以写出中文给懂中文的人看，而编程就是把想法变成程序给计算机看。先会说话后写作，学习编程也是一样，首先是要有想法，而不是先想要写什么代码。有了想法之后，就是怎么把想法通过程序实现出来。编程就是把想法变成代码的过程。互联网行业的很多创业者都有一个共同的困境，那就是“我有一个好想法，就差一个程序员了”。有了 Python，想法变程序就容易得多了。

至于学习编程的困难之处，让我们想想在学习除了母语之外另一门语言的过程吧。或许你会说“我的英语很好”“我的法语很厉害”，那么，如果用这门外语写诗呢？想想我们的古诗，要合辙押韵，还有更难的八股文，你能想象用法语写古诗、英语写八股文吗？感受到难度在哪儿了吗？古诗跟普通白话文章的区别就在于，古诗有字数限制，有韵脚的要求。而学习一门外语并且用外语写诗面临两个问题：

（1）学习一门陌生的语言。

（2）按照这门语言的语法和韵脚写诗。

编程要面临的困难也是这两个问题：

（1）不知道或不熟悉编程语言的语法和语义。

（2）不知道如何让计算机解决问题（就好像不知道怎么用法语写诗一样）。

所以，编程的困难就在于，要学习一门跟母语不一样的语言，而且为了计算机能明白，还要学习一些特定的语法和规则；同时还要用这个语言去解决问题。

明确了编程的困难，接下来就是怎样克服困难。学习编程最好的办法就是动手不断地编程，就跟学习所有的东西一样。想要做一个好的守门员，就要熟悉球性，不断练习各种基本动作（语法）、熟悉规则（编码规范）、研究团队策略（算法）等。其中最好的做法就是多练习、多实战。

最后，你需要明确编程的两个层次：

（1）找到解决问题的方法。

（2）编写出好程序。

0.2 如何写出好程序

出色大牛的程序员和菜鸟程序员的差别就在于能不能写出好程序。在我们知道了编程语言（比如 Python）的语法和语义后，怎样才能写出好程序呢？

关于什么是好程序，其定义是仁人见仁智者见智，跟行业内的朋友们聊天，特别是带团队的 CTO 们，经常会听到各种吐槽。比如：

（1）明明写的屎粑粑一样的代码，还自我感觉良好。

（2）写代码都不过脑子，上来就敲，还吹嘘自己一天写了多少行，写得再多，也只是造了更多的垃圾。

（3）瞎设计，有简单的方法不用，还自以为高明。你提醒他，他还觉得自己有个性。

（4）走路都没走稳，就想着灵活，灵活设计是给小白的吗？不是，那是给高手出招的，

小白你就踏实点。

(5) 高手是有牵绊,有顾虑的,因为被坑过……度的拿捏是艺术。

刚开始的时候,做到下面两点,那么你的程序,在你的能力范围内应该就是好的。

1. 编写程序前，一定要深思熟虑

既然编程就是解决问题的过程,只要开始编程,就意味着你要从细节开始设计,要考虑如何更好地解决问题,最终的程序是以一种最便捷的方式呈现出解决问题的思路。这就意味着,开始编程之前就要进行缜密的思考。有太多人(不只是新手)拿到项目或者要解决的问题,马上就开始编写程序。这种做法,经常会因为考虑不周,导致代码混乱,甚至中途推翻重来。

所以,开始垒代码之前,首先要做的就是坐下来,认真仔细地思考一下,怎样能更好地解决问题,找出最佳方案,再开始实施。

2. 程序要有好的可读性

在深思熟虑的基础上写的代码就是好代码吗?很多时候,问一个程序员,你的程序怎么样?他可能会回答你,能运行。能运行可能包含以下两个方面的意思。

(1) 程序能够解决问题,并且本身没问题。但这是好程序吗?其实,能运行是最低标准,是不是好程序还要看代码。通常程序员写代码都不是只给自己看的,既然是要给别人读,那么,代码的可读性就很重要,可以想象一篇没有标点、没有断句的文章,或许也能读明白,但它肯定不是好文章,也没什么价值。

可能你会问,程序能运行不就行了吗?谁没事总看?那好,如果你还没有开始工作,那么这个程序可能就是自己会看,即便这样,自己写了一个烂程序,勉强能用,等过一段时间,可能要修改一些功能,那么要做的第一件事就是先读懂之前写的烂代码,还要回想当时是怎么想的。

(2) 如果你已经在公司工作了,这种烂代码,自己看起来都费劲,你能想象分发给同事后,大家的反应吗?相信,会被“弹劾”的。

所以,经过深思熟虑之后再写出来的代码一定不会辜负你的好想法,否则,不就成了思想上的巨人,行动中的矮子了吗?

0.3 为什么选择 Python

我开始设计一种语言,使得程序员的效率更高。

——Guido van Rossum(Python 之父)[荷兰]吉多·范罗苏姆

编程就像写文章,那么,众多语言中为什么选择 Python 呢?试想作为一门陌生的、面对计算机的语言,如果这门语言更接近你的母语,而且写起来跟你说话的习惯比较类似,那么,学习编程看起来是不是也就不难了?Python 恰好就是这样一门比较容易掌握的语言。当然从某种程度来看 Python 也不是最好的,最好的是中文编程或者方言编程,哈哈。

Python 被公认为是最适合编程入门的语言之一，而且是当下最流行的语言之一，早在 2007 年和 2010 年曾两次获得编程语言排名的第一名，IEEE Spectrum 2017 年发布的报告中，Python 再度排名第一（图 0-1）。Python 语言特点简单概括如下。

（1）语法简单，容易学习和运用。

（2）面向对象编程。

（3）跨平台，具有可移植性。

（4）模块化开发，功能丰富。

（5）具有很好的扩展性。

Language Rank（语言排序）	Types（类型）	Spectrum Ranking（范围排序）
1. Python		100.0
2. C		99.7
3. Java		99.5
4. C++		97.1
5. C#		87.7
6. R		87.7
7. JavaScript		85.6
8. PHP		81.2
9. Go		75.1
10. Swift		73.7

图 0-1　IEEE Spectrum 2017 年发布的报告

Python 语言的语法简洁又独特，很短的时间内就可以写出一些看起来或实用，或很酷的程序，这在后面的学习中会深有体会。

面向对象编程最大的好处就是让程序更容易维护，增加了代码的重复利用效率。深入理解面向对象编程是程序员的必修课。

跨平台的意思就是可以在 Windows、Linux、Mac 等不同的操作系统下编写并运行 Python 程序。这样，在一个系统下编写的程序就可以移动到其他系统下运行，这就是移植。

Python 具有非常丰富的库，也就是说，很多你想实现的功能，其实都已经有人写了，你要做的就是直接拿过来使用。

扩展性是指 Python 提供了扩展接口，可以使用其他语言为 Python 编写扩展功能，比如 C/C++。当然，Python 也可以嵌入其他语言中。

有这么多的好处，Why Python？有答案了吧！

0.4 Python 的发展和应用

Python 本意是大蟒蛇，但其实来历并没有名字这么酷。Python 的创造者是 Guido van Rossum。最开始的时候，Guido 开发 ABC 语言，圣诞节闲着无聊，就想写出一个能让非编程专业的计算机使用者也能使用的编程语言，所以写了 Python，至于名字的来历，其

实是因为 Guido 是一个英国搞笑团体的粉丝，这个被称为搞笑界的披头士团体叫 Monty Python。

由于 Guido 开放源码，以至于 Python 不断壮大，甚至由于人工智能和数据挖掘等领域的大量应用，变成了网红语言。Python 从发布至今经过了一系列的演变，我们来看一下主要的几个节点。

1989 年，Guido 开始写 Python 语言的编译器。他希望这个新的叫作 Python 的语言，能符合他的理想：是一种 C 和 shell 之间、功能全面、易学易用、可拓展的语言。

1991 年，第一个 Python 编译器诞生。它是用 C 语言实现的，并能够调用 C 语言的库文件。从一出生，Python 已经具有了类、函数、异常处理、包含表和词典在内的核心数据类型以及以模块为基础的拓展系统。

1994 年，Python 1.0：1994 年 1 月增加了 lambda、map、filter and reduce。

2000 年，Python 2.0：2000 年 10 月，加入了内存回收机制，构成了现在的 Python 语言框架基础。

Python 2.4：2004 年 11 月，目前最流行的 Web 框架 Django 诞生。

2006 年 9 月 19 日发布 Python 2.5。

2008 年 10 月 1 日发布 Python 2.6。

2008 年 12 月 3 日发布 Python 3.0。

2009 年 6 月 27 日发布 Python 3.1。

2010 年 7 月 3 日发布 Python 2.7。

2011 年 2 月 20 日发布 Python 3.2。

2012 年 9 月 29 日发布 Python 3.3。

2014 年 5 月 16 日发布 Python 3.4。

2014 年，官方发表声明，对 Python 2.7 仅支持到 2020 年，并且不会再发布 2.8 版本。

2015 年 9 月 13 日发布 Python 3.5。

2016 年 12 月发布 Python 3.6。

2018 年 6 月发布 Python 3.7。

如果仔细看上面的节点肯定会发现，为什么 2008 年就发布了 Python 3.0 版本，而 2010 年又发布了 Python 2.7 版本呢？

这是因为当时 Python 3.0 发布时，就不再支持 Python 2.0 的版本，导致很多用户无法正常升级使用新版本，而当时绝大部分项目都是 Python 2.0 建立的，所以后来又发布了一个 Python 2.7 的过渡版本，之后的 2014 年声明 Python 2.7 只会支持到 2020 年，所以新手还是从 Python 3.0 入手吧。不过，作为新手也不用过于纠结版本问题，因为 2.7 和 3.0 的差别并没有达到水火不相容的地步。你掌握其中一个版本之后，如果需要用到另一版本，只需要花一点时间就能快速掌握，远比你纠结的时间少。

Python 的应用领域非常广泛，简直就是万能的。经常有人会问：“老师，Python 能干什么？有多少库？我都要学吗？”我的回答一般是：“Python 什么都能干，但是你得看你要干什么，或者对哪方面感兴趣，所有库都学的概念就好像在学字典。”

下面这几个方向是目前 Python 应用最多的地方。

- 云计算、大数据：OpenStack、Hadoop。
- Web 开发：Python 拥有众多优秀的 Web 框架，比如 Django、Flask 等。许多大型网站均为 Python 开发，如 Youtube、Dropbox、豆瓣等。
- 科学运算：NumPy、SciPy、Matplotlib、Enthought librarys、pandas。
- 人工智能：Scikits-learn、Tensorflow。
- 系统运维：运维人员必备语言，自动化运维必不可少的工具。
- 金融：量化交易、金融分析。在金融工程领域，Python 不但在用，且用得最多，而且重要性逐年提高。主要原因是作为动态语言的 Python，语言结构清晰简单，库丰富，成熟稳定，科学计算和统计分析都很专业，生产效率远远高于 C、C++、Java，尤其擅长策略回测。
- 图形 GUI：PyQT、wxPython、Tkinter。
- 游戏开发：Pygame、pyglet、cocos2d-python。

不过，看起来万能的语言，倒也不是什么都用 Python 去做，实际情况还是要选择适合的语言做对应的工作。比如 Python 的运行速度就不是最快的，若有速度要求，用 C++ 改写关键部分吧。不过对于用户而言，机器上的运行速度是可以忽略的，用户几乎感觉不出来这种速度的差异。

0.5 一些建议

每个人的学习方法都不一样，不过对于初学者，我有一些建议，希望对你的学习经历能起到一些帮助，少走些弯路。

1. 保持好奇心

> 你内心肯定有着某种火焰，能把你和其他人区别开来。
>
> ——［南非］约翰·马克斯韦尔·库切《青春》

我上小学时，无意间看到一本中学的物理书，上面有些电路图，有些就是电池、开关、灯泡的串并联，虽然不明白物理的专业名词，也不懂原理，但看着图觉得很好玩(图 0-2)，读了书上的说明，我就可以自己按书上的例子做出来，这时就会觉得很有成就感。当然，有时也会有危险，比如有一次接了一个电铃，在插头插入插座的一瞬间，电线整个烧煳了，还好没有电到我。

学习编程也会有很多好玩的东西，有了想法就试着去做，细心大胆地试，编程安全得多，至少不太可能写错一个程序就让你的计算机报废。

对于新生事物，我们总是心怀好奇心，好奇心也是能够让你轻松投入又没有负担地学习的原动力。

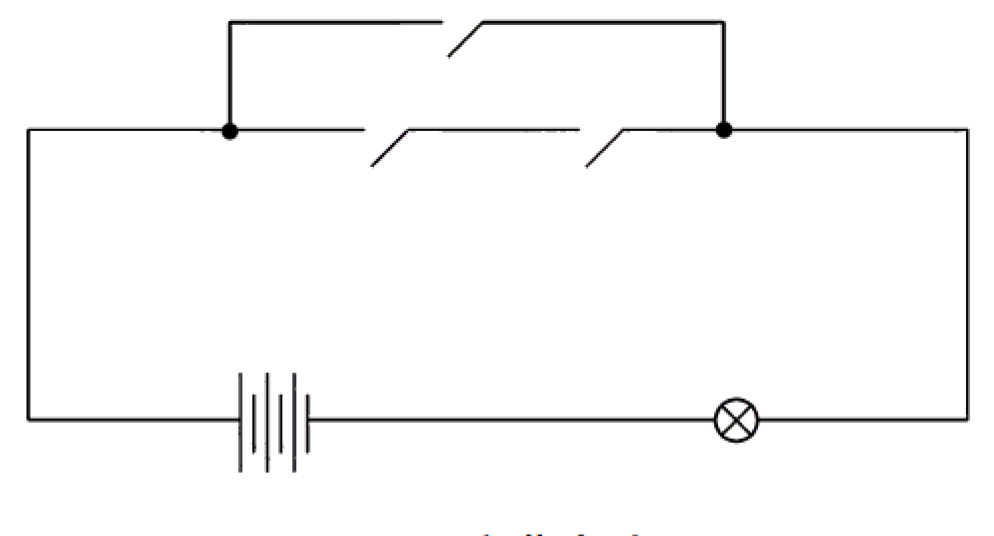


图 0-2 启蒙电路图

2. 不要感觉枯燥

有人觉得学了编程做程序员，工作就注定很枯燥了，no no no……编程和写作一样，都是在搞创作，这是一个从无到有的过程，从能用到好用再到高效的创作过程。编程是门艺术，程序的设计和度的拿捏是需要功力的。程序员是最有创造力的一个群体，所以用心去感受吧。

3. 做笔记，但是不要抄代码

学习过程中碰到新的知识点、新的模块、新的技巧，可以多做学习笔记，最好可以开个博客，或者找一个在线的大本营，记录并跟更多人分享你的学习过程和经验，跟更多的人交流总是可以获得更大的帮助。做学习笔记，但是不要抄代码。你把网上看到的代码复制过来放到自己的博客或者空间想要留着以后看或者觉得放在那儿就算会了，这跟抄代码一样，都是不对的。所有抄代码的行为一点益处也没有，最好的办法是自己做一遍，再把自己的代码和心得保存下来。

4. 多动手，不要只是看

编程不是做阅读理解，看得再多也是没用的，一定要多动手实践。对于初学者、小白、零基础，更是如此。再简单的程序也要动手操作一下，不动手，可能一个空格、一个符号都是陷阱。当然，动手了也会碰到各种陷阱，但这个过程就是你成长的过程。

5. 细心一点，知其然，知其所以然

因为有太多次类似的经历，不得不说一下。经常有人会拿着程序来找我问："邹老师，你看我的程序和你的例子一样，怎么结果就不一样呢?""怎么你的能运行，我的就报错呢?"每当这种时候我是最崩溃的，我的第一个问题都是："一样吗？你再看看，你确定一样吗?"往往这时自己检查后就会发现问题所在。其实类似的问题，马虎只是一个方面，最主要是因为当你在照着别人的代码敲时，要么根本不理解在做什么，只是机械地一个一个跟着敲，最后可能差个空格，多个符号等；要么是运行环境根本不对，比如需要的库没有安装，环境本身有问题等。所以，细心一点，重要的是知其然更要知其所以然。

6. 多寻求互联网的帮助

互联网如此发达的今天，你碰到的问题可能别人早就遇到过，已经提供了解决办法。那么，有问题时多到网络上跟大家交流是个不错的办法，比如各种技术论坛、问答组、交流群。但是，这种求助不包括代码写不出来去网上求代码，这跟上学抄作业一样，坚决不能。

7. 提问的艺术

说到了寻求帮助，在学习过程中碰到问题是再正常不过了，不论初学者还是高手都会遇到问题。但是，大家寻求帮助的方法却有很多错误。Eric Raymond 2001 年发表的《提问的艺术：如何快速获得答案》一文，通过对好问题和坏问题的比较，结合自己的经验，对如何能在互联网快速获取到帮助，做了详细的阐述。希望读者朋友有时间一定看一看。这里我简单地说几点。

(1) 作为能回答问题的高手，只喜欢值得思考的问题，相对来说，有些问题本身就是垃圾，不动脑筋的人是不会得到帮助的。所以，在提问之前，首先尝试自己找答案，手册、文献、图书、搜索引擎等都是你的工具。

(2) 提出问题时，有偿提问或免费社区都可以选择。首先要说明在此之前你干了些什么，这样能证明你不是一个想不劳而获的伸手党。问题的语法要正确，用词要准确，因为没人愿意回答明显有错别字或关键字的代码，比如 for 写成了 fro 等。问题描述准确但不啰嗦，描述不清就提供代码或者截图。

(3) 别问本该自己解决的问题，那是你自己学习的过程，不是别人的义务；去除无意义的问题，别问 1 加 1 为什么等于 2；懂得感恩，无论提问前还是得到答案后。

0.6 多平台搭建 Python 开发环境

"工欲善其事，必先利其器。"学习编程首先需要具备编程的环境，并且能够确切地知道编程环境的作用，因为这个环境不是指鸟语花香的书房。我们需要做的准备工作如下。



搭建 Python 环境

(1) 一颗好奇的心。

(2) 一台能上网的计算机。

下面我们就带着好奇心开启编程之旅吧。

0.6.1 什么是开发环境

我第一次接触编程语言时什么都不懂，确切地说，那时候是在电视上看到的"C 语言教学"课程，正好那节课讲的是用"*"在屏幕上排列出一个五角星。出于好奇心，对于一个在大多数家庭还没有计算机的年代，如果会这件事，我觉得非常酷。于是我很认真地记下了过程和代码，之后，我把那段代码背得烂熟于心，希望有一天可以在人前炫耀。

当终于有了一个机会可以在计算机上敲代码时，我迫不及待地把背下来的代码演示给

同学看。但是，却以失败告终，至于原因，我当时的解释是，学校的计算机太垃圾，哈哈。

直到后来电视重播了那套“C语言教学”课程，我才知道，第一课讲的不是代码，而是开发环境，教的是搭建C语言开发环境。回想我之前编程的环境，仅仅是在记事本里输入了我的代码，哪里有什么环境。

那么，什么是开发环境呢？简单来说，就是计算机中的一个“翻译”，它能把你写的程序解释给计算机去执行，还能够把计算机执行的结果翻译成你能懂的信息返回给你。

Python 最基本的开发环境就是在计算机上安装一个叫 Python 的软件，这样在 Python 软件提供的环境里执行 Python 指令时，计算机就能明白你要做什么了。

0.6.2 获得 Python 安装包

登录 Python 的官方网站 www. python. org 即可获取 Python 安装包。第一次接触到这个网站的顺便认识一下吧，这里不仅可以下载软件，也是 Python 编程过程中获得第一手信息的地方，这里不仅有 Python 软件包，还有 Python 的详细使用手册和 Python 的模块仓库，除了安装 Python 时自带的模块以外，这里还有很多第三方开发的模块。

先来找 Python 的安装包。打开 www. python. org（图 0-3）网站，单击主页中的 Download 按钮，找到合适的版本下载即可。找到软件包时，你会看到很多版本，有针对不同系统的，也有同一个系统的不同版本，首先根据你的操作系统选择对应版本，然后会发现有 Python 2. * 和 Python 3. * ，在本书中，我们采用 Python 3. 6. 2 进行演示，这并不是说 Python 2. * 就是旧的、已经淘汰的版本，恰恰在这本书创作时，Python 2 还是国内很多网站的主流，这是有历史原因的。

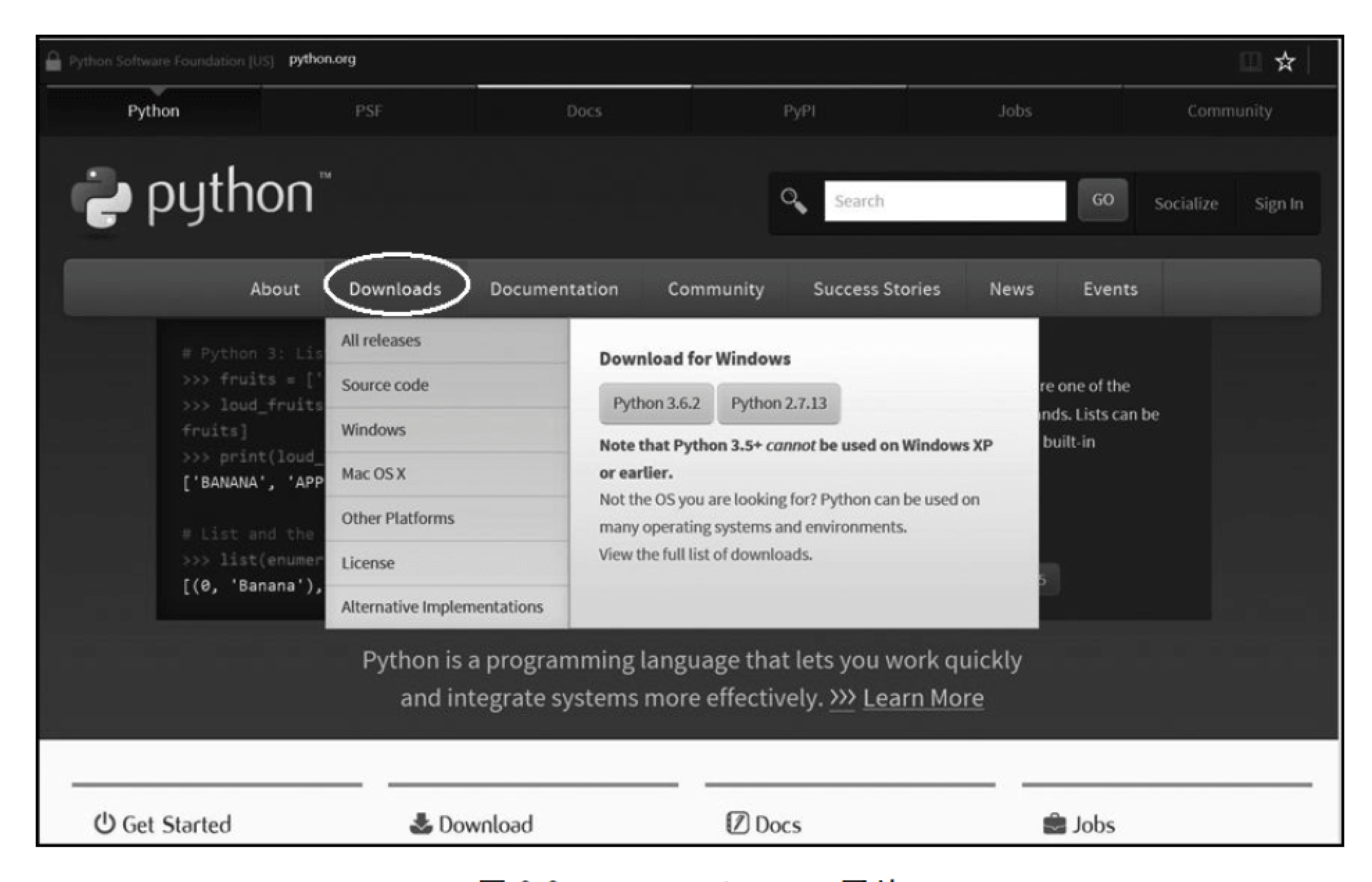


图 0-3 www. python. org 网站

作为 Python 程序员经常会自黑 Python 2 和 Python 3 这两门语言，实际并没有那么严重。虽然有些区别，但在初学阶段不需要太纠结（后面会涉及一些比较明显的差异）。

0.6.3 安装 Python

注意：不要在一个系统上同时安装 Python 2 和 Python 3，除非你清楚地知道怎么区分它们。

下载完成后就可以开始安装了。在 Windows、Mac 和 Linux 系统下的安装过程如下。

1）在 Windows 系统下安装

开始运行安装程序后，本书没有图示的步骤，只要单击"下一步"按钮即可，到图 0-4 所示这一步时需要注意，粗箭头所指的 Add Python 3.6 to PATH，需要勾选上，如果没勾选，安装之后会导致无法运行 Python。



图 0-4　安装选项设置

另外，关于 Customize installation，你可以打开看看（见图 0-5），作为定制部分，一般默认即可。在没搞清楚都是什么功能时，建议全部安装，其中有个 pip，在本章我们就会用到。看到图 0-6 就说明安装成功了。

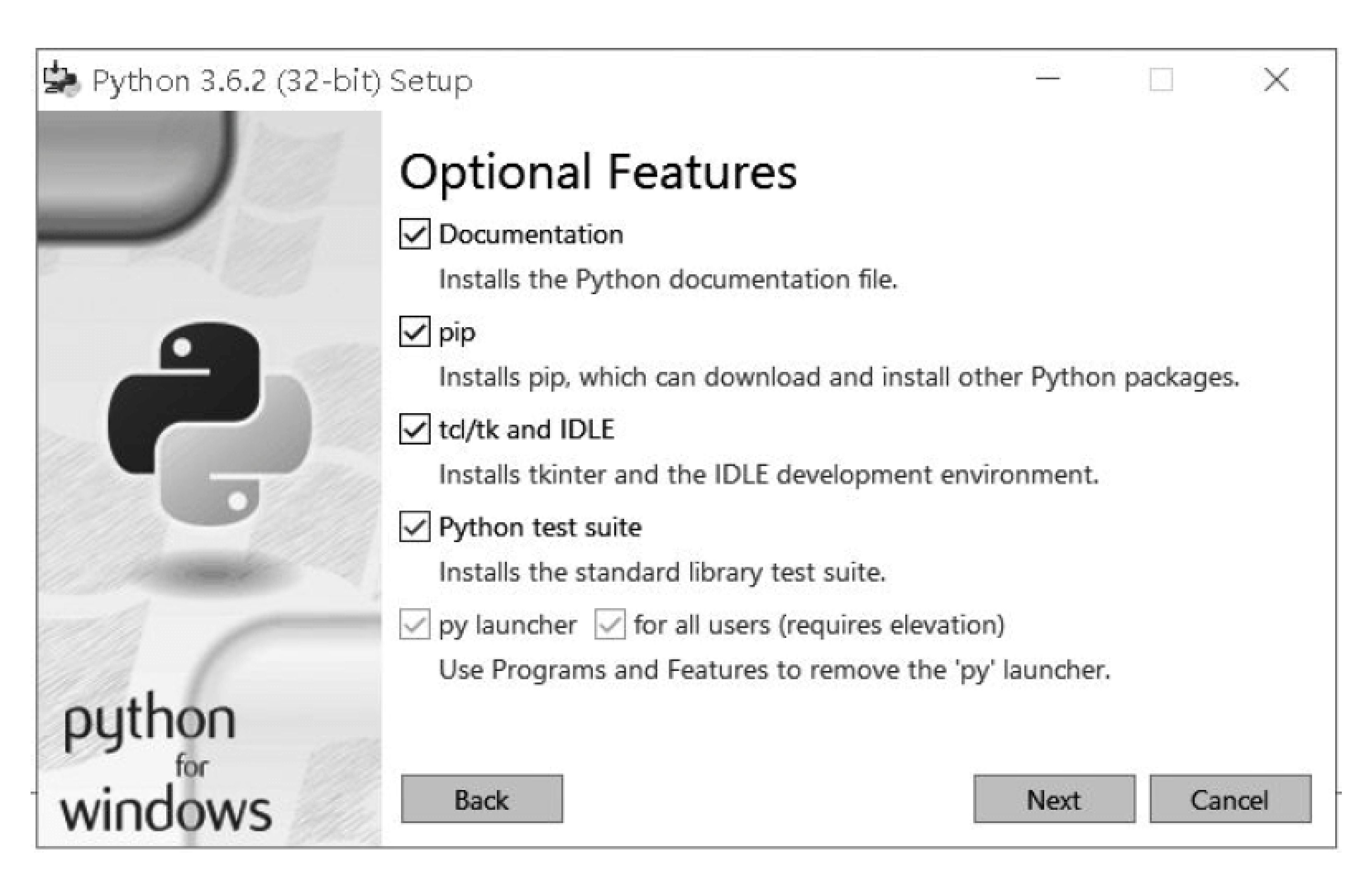


图 0-5　定制功能界面



图 0-6　安装成功

2）在 Mac 系统下安装

在 Mac 系统下安装要简单得多，只要按提示单击“下一步”按钮即可，此处省略图示讲解。

3）在 Linux 系统下安装

若你是 Linux 系统，恭喜你，大部分主流 Linux 都自带了 Python 2，有的发行版已经自带了 Python 3，采用默认版本即可。

0.7 命令行模式及 Python 的第一次运行

软件安装好之后，就可以检验一下成果了。由于 Python 是跨平台的，分别在不同的系统中安装了 Python，就好像在 iPhone 和安卓上安装了同一款软件一样。手机系统不一样，但软件一样，其使用也是一样的，所以不同平台 Python 开发的过程是一样的，只是有些外在形式可能存在差异。

运行 Python 之前，你要明确一点，在计算机中，除了熟悉的图形界面外，还可以通过字符界面控制计算机，也就是命令行模式。是的，就像黑客电影一样。你不要以为既然有了 Windows 图形化的窗口系统，谁还用字符界面。其实，作为程序员，非常有必要了解命令行模式，不用多，基本的操作暂时就够了。因为命令行模式相对于图形更直接、更纯粹，相对来说工作效率更高。

(1) 先来看一下 Windows 用户吧。一般情况下，Windows 用户对字符界面会比较陌生，作为第一次运行，你可以用最经典的 cmd(命令提示符)。以 Windows 10 为例，你可以直接搜索 cmd(见图 0-7)就能找到了，运行界面如图 0-8 所示。Windows 中还有一种更强大的命令模式，就是 PowerShell，你也可以直接搜索使用(见图 0-9)。基本使用方法是一样的，只是 PowerShell 支持更多的命令，样子看起来也比 cmd 高级了许多，并且可以自己进行一些配置调整。建议调整为黑背景白字。Windows 用户推荐使用 PowerShell，具体运行界面如图 0-10所示。



图 0-7　搜索启动 cmd

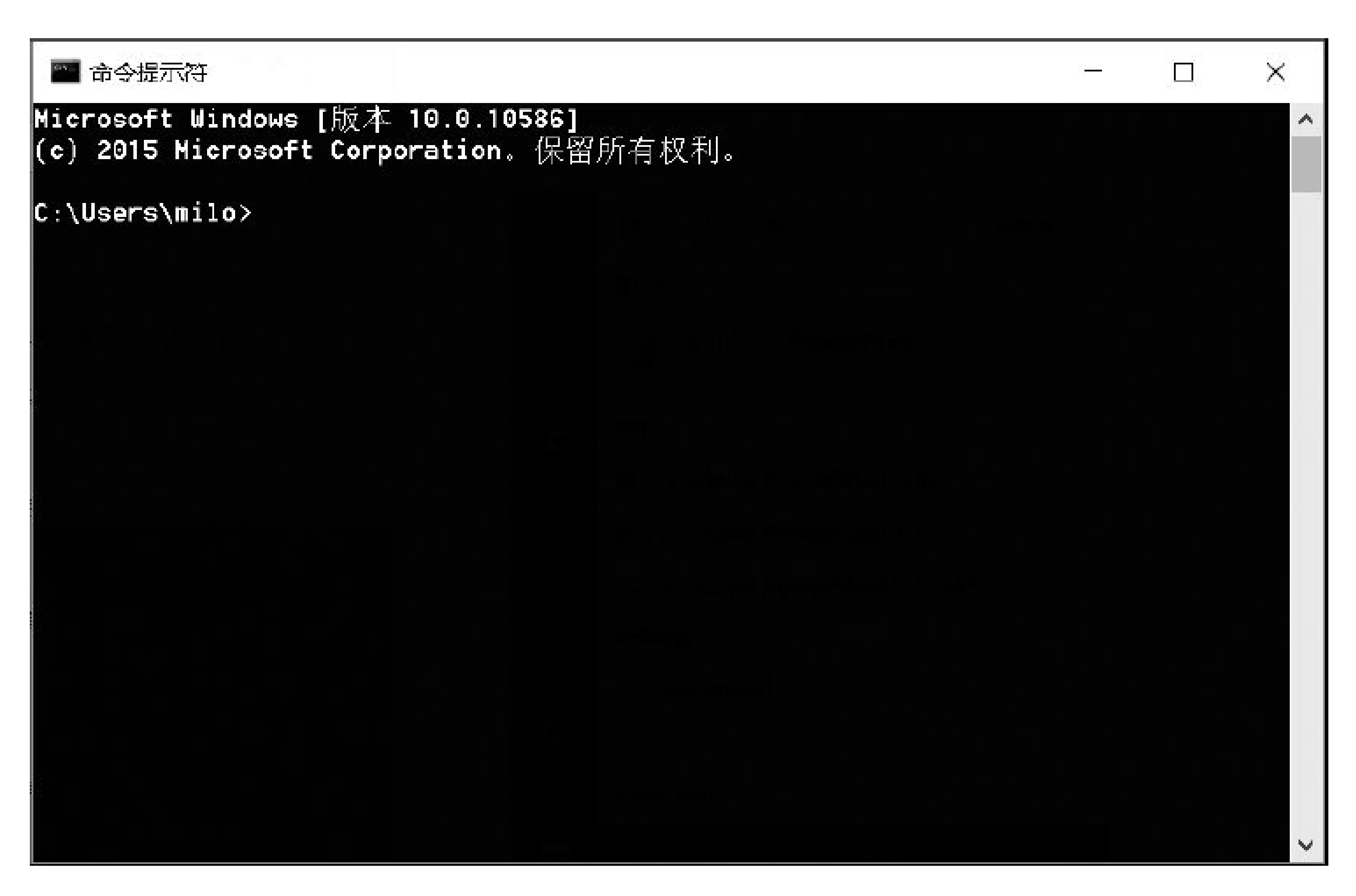


图 0-8　cmd 运行界面

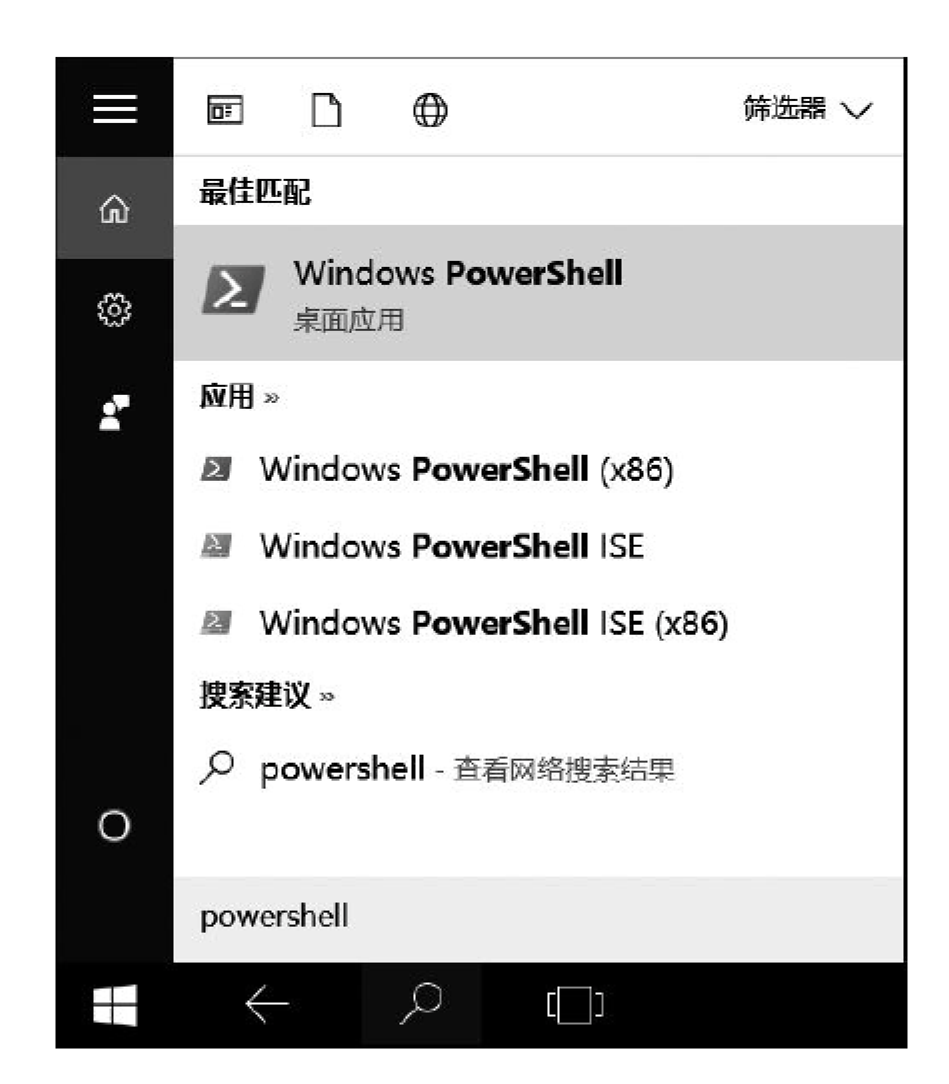


图 0-9　搜索启动 PowerShell

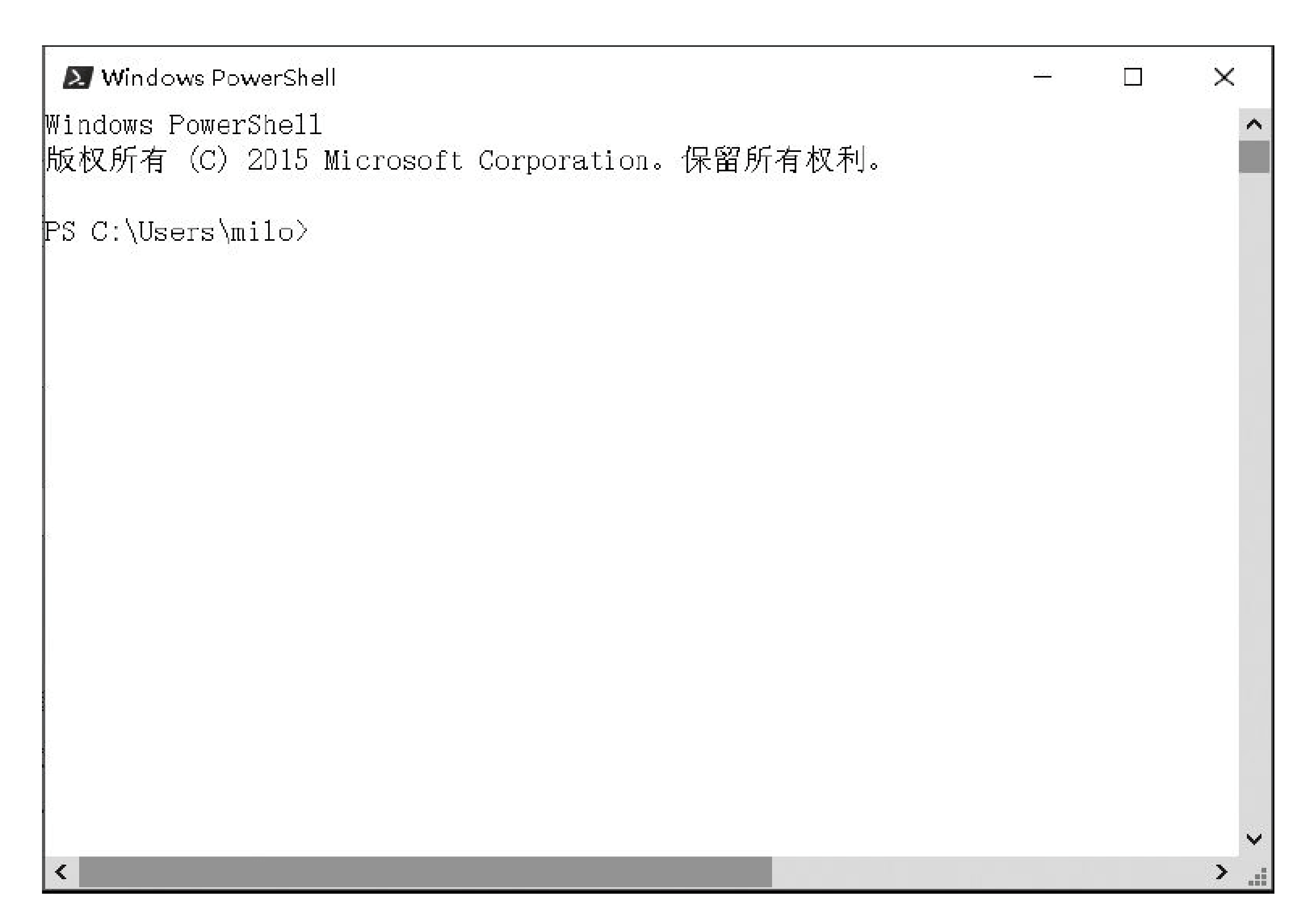


图 0-10　PowerShell 运行界面

(2) Mac 用户也一样，再炫的图形界面也不如字符终端实在。Mac 用户只需要打开 terminal(见图 0-11)终端即可，terminal 原始位置是 finder→工具栏→应用程序→实用工具→终端。

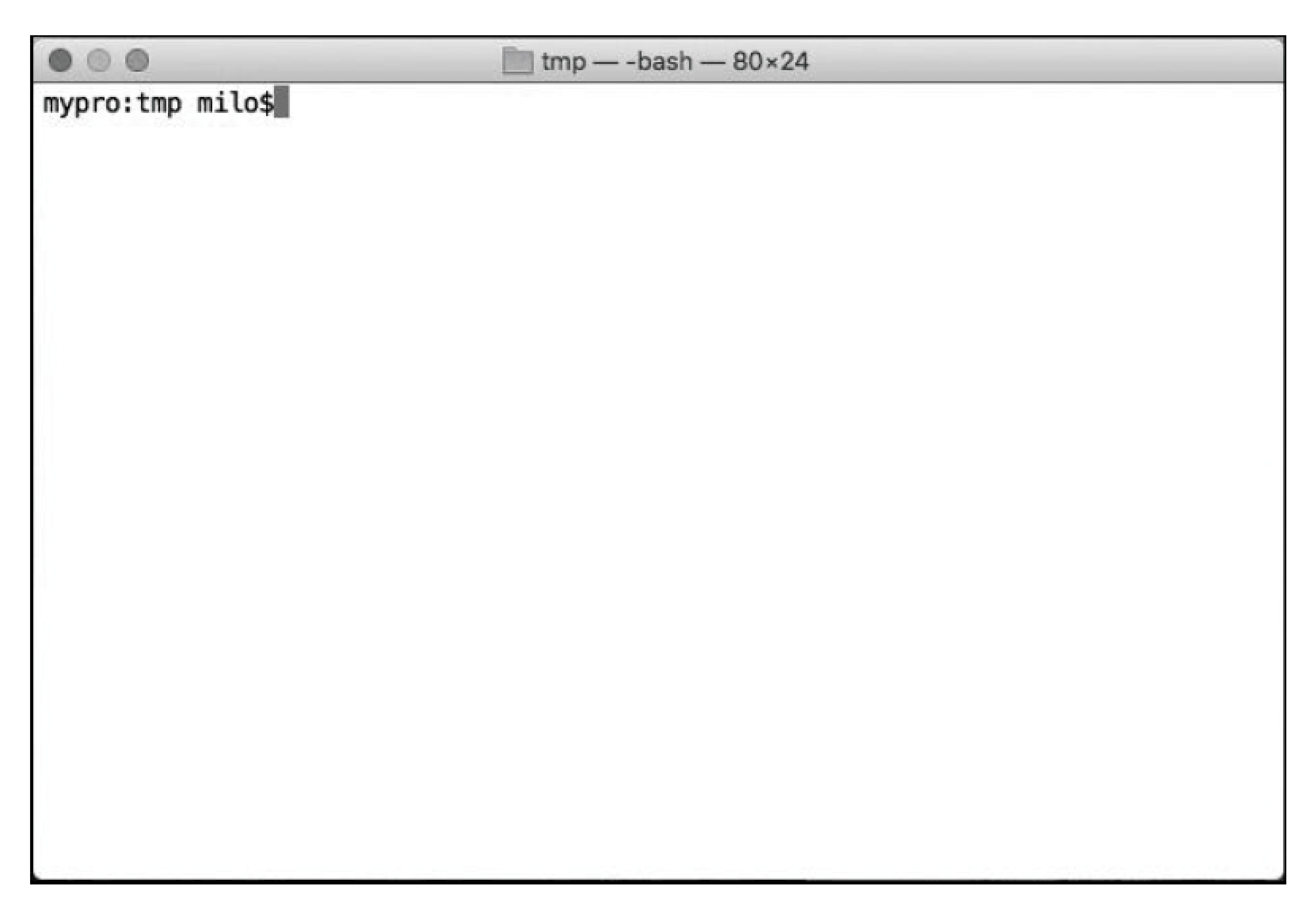


图 0-11　terminal

(3) 至于 Linux 用户，我想说如果你是 Linux 用户难道还需要看我介绍吗？作为一个 Linux 忠实粉丝，直接启动终端吧(见图 0-12)。

以上就是用户群体最多的三个操作系统关于字符终端的启动方式，这时候你可以执行一些系统命令。输入命令，然后按 Enter 键执行。

以 Windows 为例，你可以看一下自己现在处于系统的什么位置(见图 0-13)，你当前所在目录下有哪些文件(见图 0-14 和图 0-15)。

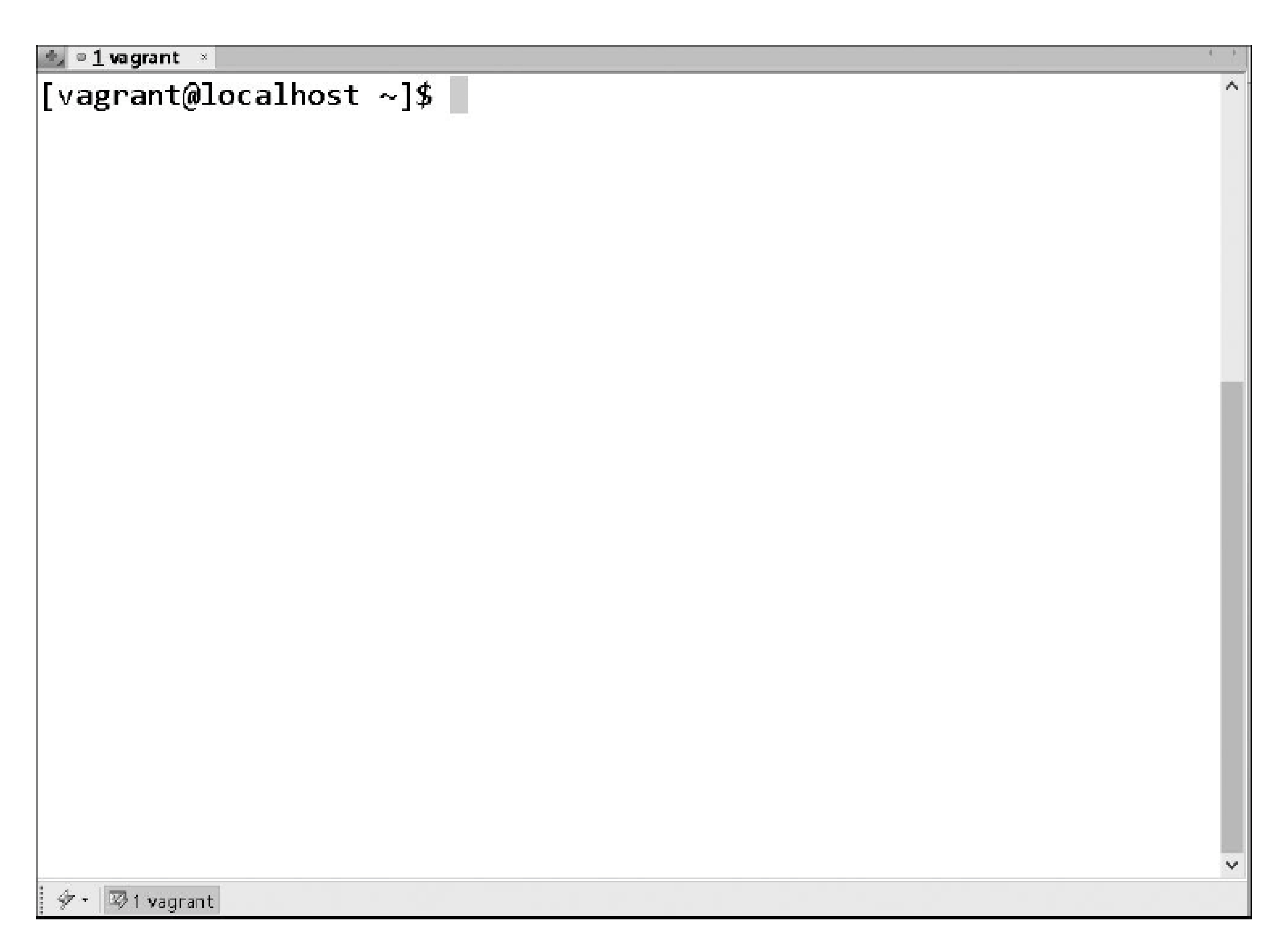


图 0-12　shell

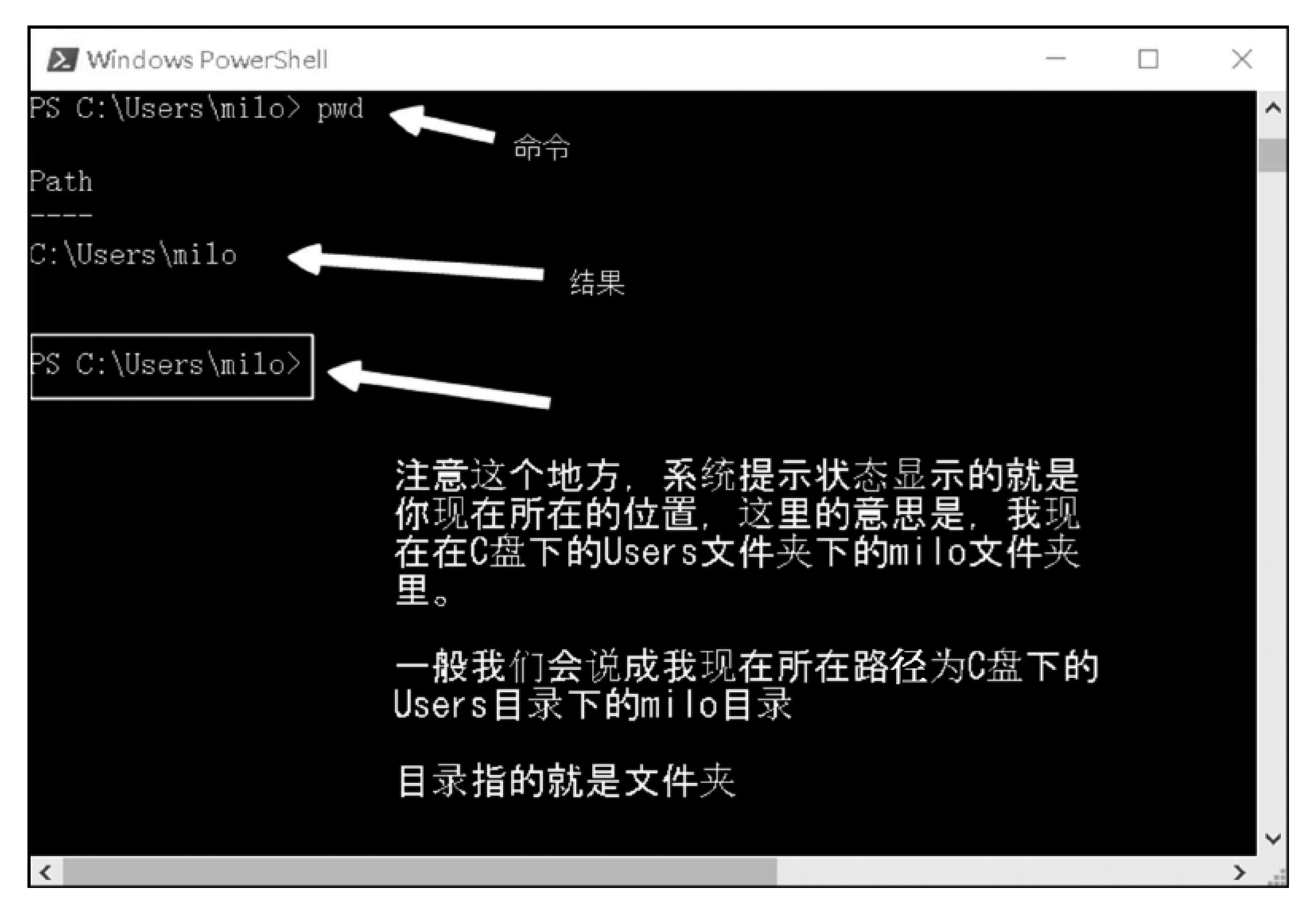


图 0-13　显示当前所在路径

如果屏幕显示满了，可以执行清屏指令 clear 或者按 Ctrl+L 组合键（Windows 的 cmd 不支持，PowerShell 可以）。

适应了命令行模式后（这时你所输入的命令都是直接下达给计算机的），接下来我们进入 Python 语言的环境下执行 Python 指令。直接在字符终端输入 python，回车即可看到如图 0-16 所示界面，明显的标志是有“>>>”的提示符。看到这样的界面，就表示已经在 Python 语言环境中了，你可以输入第一个 Python 指令（确切地说是执行 Python 的 print() 方法）在屏幕上输出你的名字（见图 0-16）。

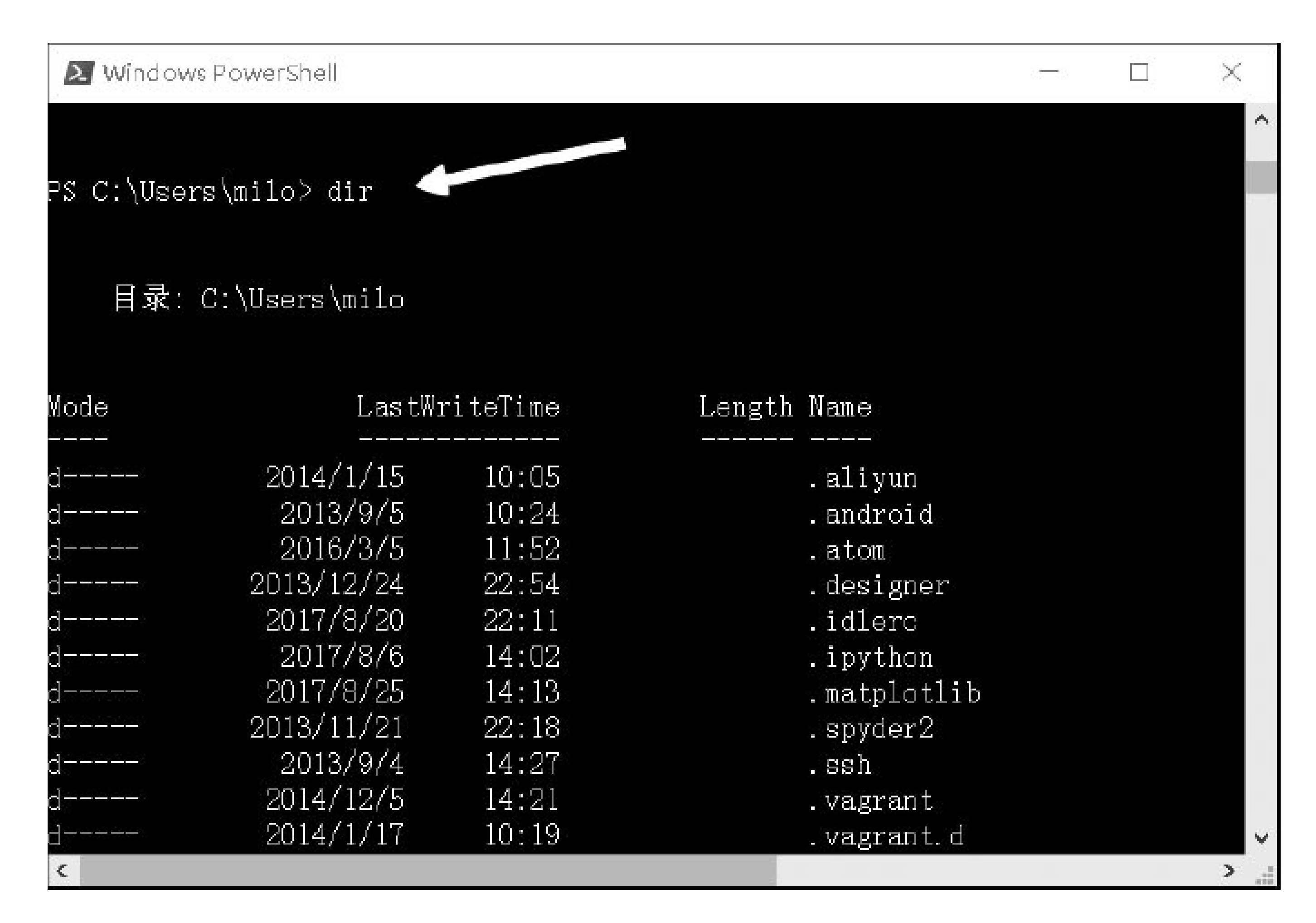


图 0-14　显示当前目录下的文件

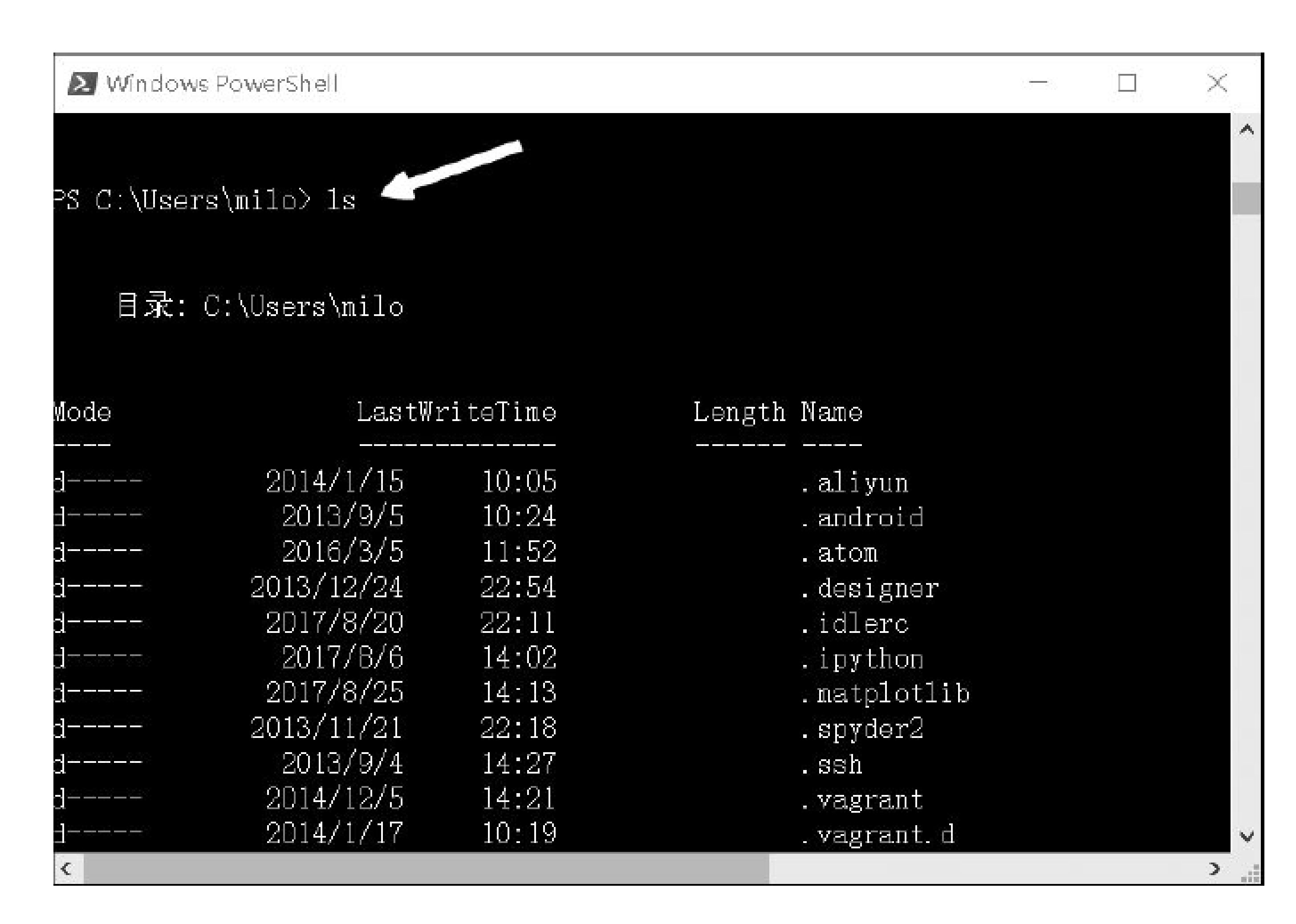


图 0-15　PowerShell 也支持 ls 这样的 Linux 命令



图 0-16　Python 交互环境

注意：此时需要明确的是，你是在系统环境中还是在 Python 环境中，因为在系统环境中只能执行系统命令，不能执行 Python 指令。同样，在 Python 环境下也不能直接执行系统命令。接下来的课程中有些指令是系统命令而不是 Python 的指令，需要进行区分。

最后，我们执行完 Python 指令，可以执行 exit()指令退出 Python 环境(见图 0-17)，这就好像你在系统中玩完游戏退出，回到系统是一样的。

```
Windows PowerShell
PS C:\Users\milo> python
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:14:34) [MSC v.1900 32 bit (Inte
Type "help", "copyright", "credits" or "license" for more information.
>>> print("my name is milo !")
my name is milo !
>>> exit()
PS C:\Users\milo>
```

图 0-17　exit()

到此，就完成了与 Python 的第一次对话，我们下达了在屏幕输出自己名字的指令，Python 解释给计算机之后，在屏幕上打印出了指定的内容。

本书的课程基于 Windows 系统进行讲解，因为 Python 本身支持跨平台，所以，实际上在哪个系统上操作都是一样的，因为都是在 Python 环境下开发，用什么系统主要取决于你要做什么事。

重点提示

在这一章中，你要掌握的内容：

(1) 意识上跨过编程的门槛。

(2) 能顺利搭建 Python 环境，注意 PATH。

(3) 熟悉系统的命令行模式，了解基本操作。

(4) 熟悉 Python 的交互模式，并在交互模式下执行 Python 指令。

动动手

(1) 浏览 Python 官网 https://www.python.org，下载跟自己系统对应版本的安装包，找到对应版本的 Python 文档并且浏览。

(2) 搭建 Python 环境并测试。

(3) 在系统中创建一个用来保存代码的文件夹 My Python Code。

(4) 在命令行模式下找到并进入 My Python Code 文件夹。

(5) 在 Python 解释器中执行 print()输出一些个人信息，比如身高、体重等。

(6) 在解释器中直接做一些数学运算，比如 1+1、3−2、5×6、10/2。

(7) 计算个人的 BMI(即身体质量指数)，其值为体重(kg)/[身高(m)]^2，成人正常范围是 18.5～23.9，低则瘦，高则胖。

第 1 章 开始编程

上一章,我们已经搭建好了开发环境,接下来是否该学习程序最基本的组成元素和语法结构了呢?作为一本骨骼清奇的编程书,怎么能这么俗套呢?有了编程环境,我们现在就开始编程吧,老师都是看了一节电视教学就编五角星程序,你也来编程吧,别等着我罗列知识点了,干活吧!

从需求出发,把想法变成代码,熟悉并遵循编码规范,这就是编程的三要素(我定义的)。掌握这三个要素,就算入门了,后面的学习,无非就是:

(1) 根据不同的需求,分析出解决办法。

(2) 学习各种语法以便支撑你的想法变成程序。

(3) 编写代码时遵循通用规则。



开始编程吧

1.1 第一个程序的诞生

现在你已经了解了字符终端和 Python 的运行环境,就已经算是有两把刷子了,我们就来编程吧。不要顾虑太多,开始编程只需要一个理由,就是我要做什么,现在要做的就是画一个五角星。

1.1.1 编程动机

编程之前先不要急着考虑程序怎么写,回想一下前面讲的,写文章之前,首先是有一个想法,然后再找合适的文字把它变成文章。编程也一样,如果我们的目标是要画一个五角星,那么你首先想到的是什么?是还不了解语法的程序代码?还是程序运行的效率?都不是,应该是脑子里的五角星的形状,如果在纸上画,起笔顺序应该也想好了吧!

把在纸上画五角星这个过程变成程序,让计算机来画,画出第一个,就可以画出满天繁星,这就是你的编程动机。不管是谁,能用最简单便捷的方法实现自己的想法,就是好程序。

1.1.2 神奇的导入: import

只靠脑子想没有用,因为现在还是小白,脑子里没东西,所以,有了想法之后,就要通过编程语言把想法变成程序,这时就要充电学习了。

前面我们提到 Python 中有很多模块(也可以叫库),它们可以做很多事,现在我们想画

图，那就找一个能画图的库，这比生涩的语法要容易得多，甚至不需要明白关键代码怎么写。这些库通常都有帮助文档或手册，只是拿来按说明使用就可以了。在 Python 中使用这些模块的第一步就是通过 import 指令导入模块，语法如下：

```
>>>import 模块/库名字
```

先来试着导入第一个：

```
>>>import this
```

我们并没有使用 print()向屏幕输出，但是我们看到了一段话(见图 1-1)，这段话就是由模块 this 实现的效果，内容是著名的 Python 武功心法——Python 之禅。它是 Python 程序设计的哲学。如果时间允许，建议逐条看一看，学习过一段时间之后再回来看这段话会有更深的体会。

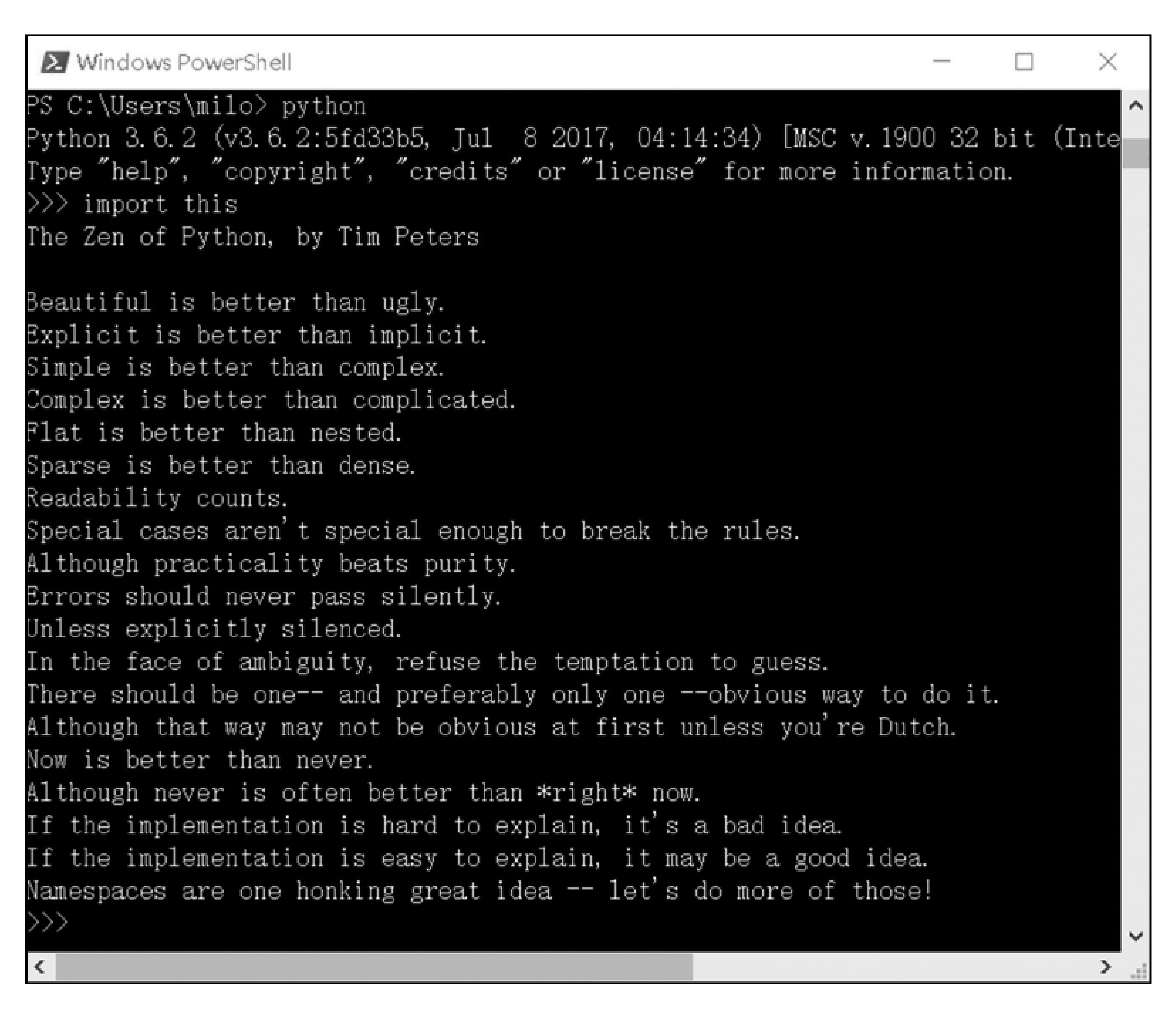


图 1-1　Python 之禅

Python 之禅 by Tim Peters

优美胜于丑陋（Python 以编写优美的代码为目标）。

明了胜于晦涩（优美的代码应当是明了的，命名规范，风格相似）。

简洁胜于复杂（优美的代码应当是简洁的，不要有复杂的内部实现）。

复杂胜于凌乱（如果复杂不可避免，那代码间也不能有难懂的关系，要保持接口简洁）。

扁平胜于嵌套（优美的代码应当是扁平的，不能有太多的嵌套）。

间隔胜于紧凑（优美的代码有适当的间隔，不要奢望一行代码解决问题）。

可读性很重要（优美的代码是可读的）。
即便假借特例的实用性之名，也不可违背这些规则（这些规则至高无上）。
不要包容所有错误，除非你确定需要这样做（精准地捕获异常，不写 except：pass 风格的代码）。
当存在多种可能，不要尝试去猜测。
而是尽量找一种，最好是唯一一种明显的解决方案（如果不确定，就用穷举法）。
虽然这并不容易，因为你不是 Python 之父（这里的 Dutch 是指 Guido ）。
做也许好过不做，但不假思索就动手还不如不做（动手之前要细思量）。
如果你无法向人描述你的方案，那肯定不是一个好方案；反之亦然（方案测评标准）。
命名空间是一种绝妙的理念，我们应当多加利用（倡导与号召）。

好了，可能在屏幕上看到一段话并没有多神奇，那么，你在计算机联网的前提下试着执行下面这条指令：

```
>>>import antigravity
```

怎么样？你只是执行了一条指令，计算机就自动打开了浏览器，并且访问到一个网站，看到了这幅漫画（见图 1-2）。就像漫画上说的一切都是这么简单。

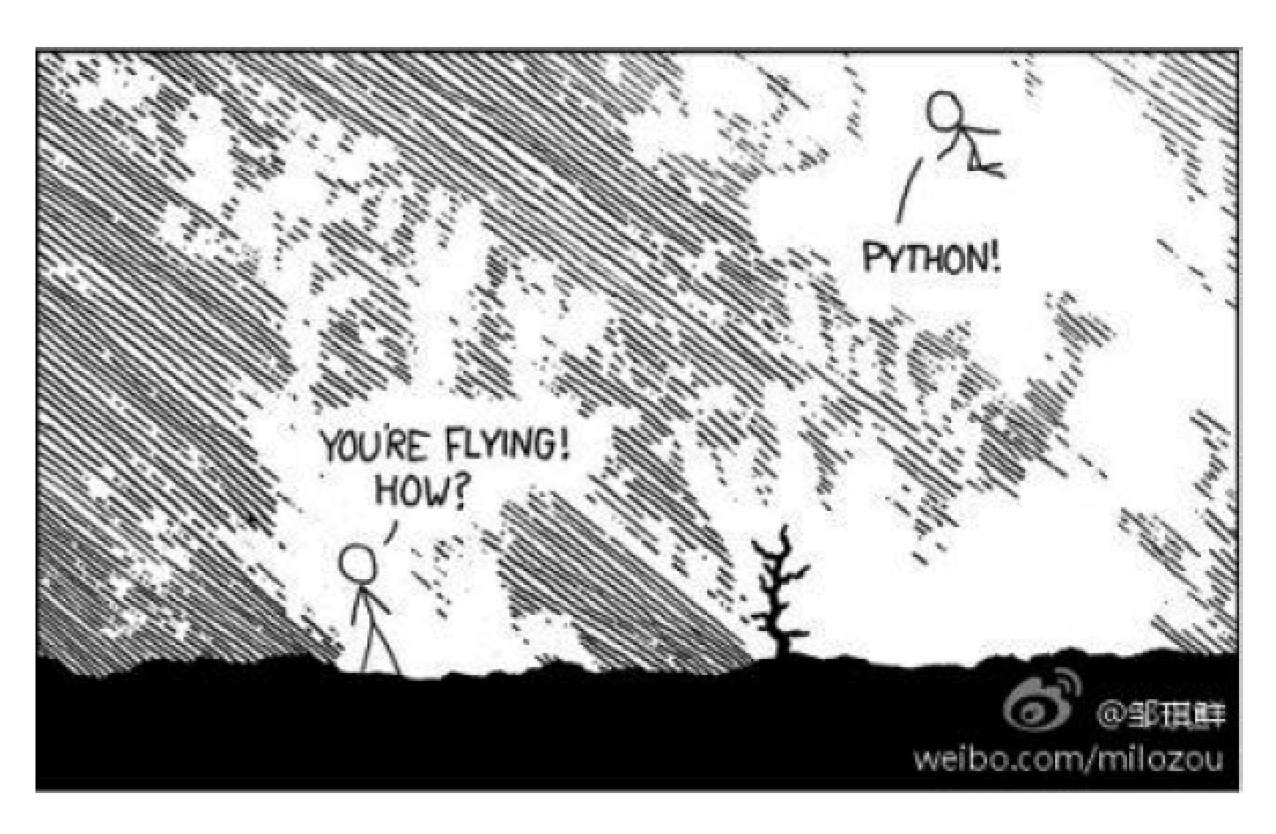


图 1-2　import antigravity 效果图

1.1.3　画一个五角星

知道了 Python 有很多神奇的模块，我们就可以开始画图了。这里要用到一个叫 turtle 的模块，这个模块用起来就像它的名字一样，我们控制一只小海龟在沙滩上爬行，通过指挥它的运动轨迹来画图。想一下画五角星运笔的顺序，每条边多长，每个角多少度。我的想法是先画一条直线，然后转 144°，再画一条直线，再转 144°，直到把整个五角星画完。

想法确定后就可以开始了，首先需要导入 turtle 模块：

```
>>>import turtle
```

这时你还看不到任何效果，接下来开始学习指挥它。通常模块中会提供很多方法，这

里我们指挥 turtle 画五角星只需要两个方法：画直线和转向。直线方法是 forward()，转向方法是 right()或 left()。调用方法是点记法，比如，直行就写作 turtle. forward(200)，点记法在这里就是用来表明调用 turtle 里面的 forward 方法。括号中的数字代表前进的距离，单位是像素(像素就是屏幕显示的最小的点)，这条指令的意思就是直行 200 像素。而转向写作 turtle. right(144)，括号中的数学是转向的角度。我们一条一条地下达指令让 Python 去解释执行，你会看到每条命令的效果(见图 1-3)。

【五角星指令】

```
>>>import turtle
>>>turtle.forward(200)
>>>turtle.right(144)
>>>turtle.forward(200)
>>>turtle.right(144)
>>>turtle.forward(200)
>>>turtle.right(144)
>>>turtle.forward(200)
>>>turtle.right(144)
>>>turtle.forward(200)
>>>turtle.right(144)
```

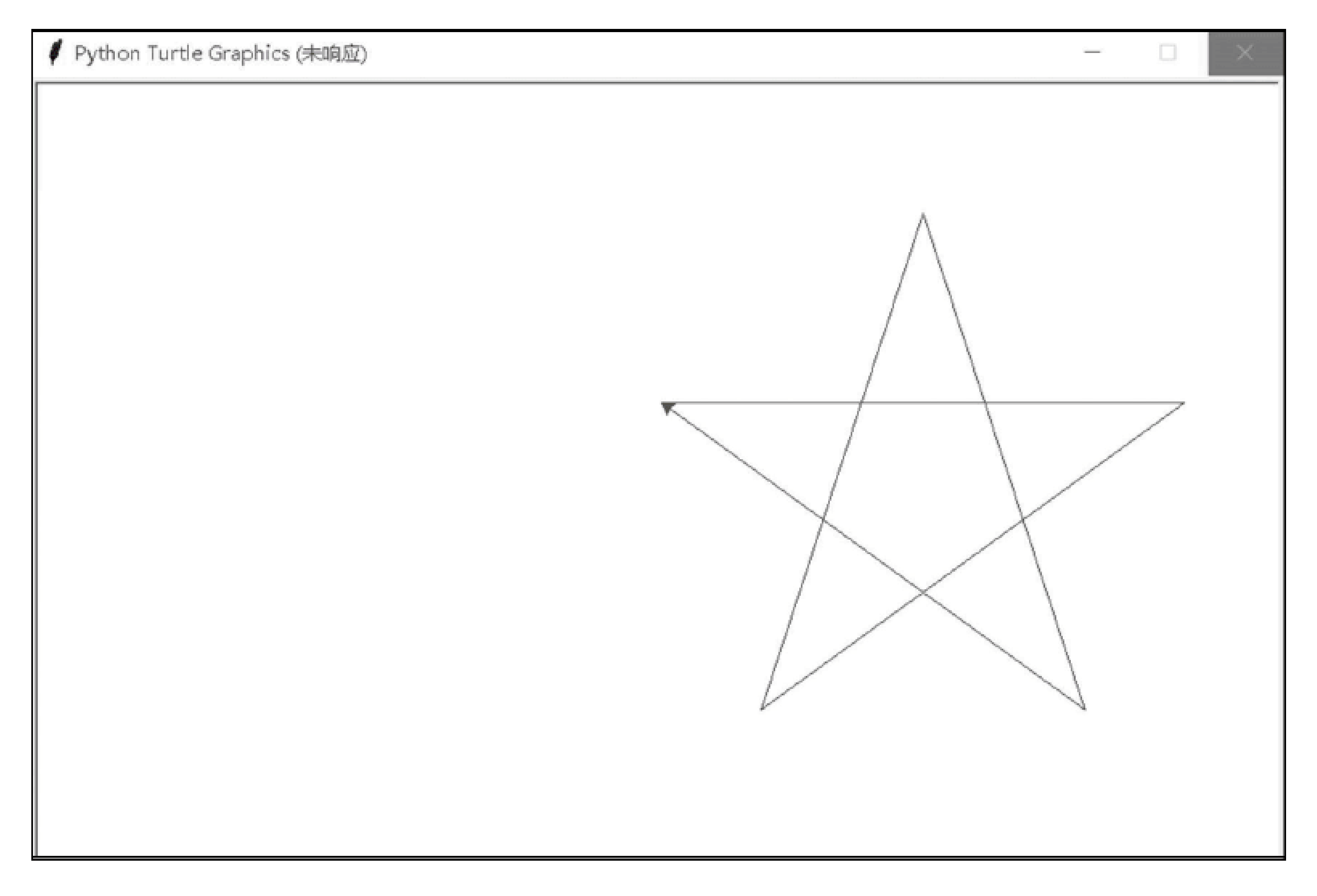


图 1-3　turtle 画的五角星

注意：随着学习的深入，指令会逐渐增加，也会越来越复杂，建议在对不熟悉的新功能进行测试或者出问题进行筛查时，逐条执行，这是测试的一种方法，能够方便你找出问题所在，也是交互模式的意义所在。

1.1.4　Python 对话

绘制五角星的过程就是一次交互式会话，我们下达一条指令，计算机回馈一个执行效果。这是因为 Python 是解释型语言，这个特征使学习和试错更方便(所谓的解释型就是程序在执行过程中，会通过 Python 解释器每次读取一行 Python 代码并逐行解释)。这意味



着，我们可以在学习或者尝试新功能时，先在 PythonShell 环境中实验代码，可以很快、很直观地看到效果，成功或者失败。这个功能非常有用，这也使 Python 更易于探讨和学习。我极力建议在学习新功能时，先通过交互式会话进行测试。

1.1.5 编写程序

现在，你已经可以通过 Python 对计算机下达指令了，但是看起来这还不像程序。那什么是程序呢？程序就是指令的集合。把你的指令保存到一个文件里，这个文件就是程序，编程其实就是在编这个文件里的内容。所以，现在要做的就是把能画出五角星的所有指令，按照每行一条的形式保存到一个文件里，作为 Python 程序，这个文件的后缀名必须是 .py，然后文件命名尽量简单、易懂，wujiaoxing.py 就不错。若无必要，不要用中文，也不要用有特殊意义的名字，比如这个画图程序会用到 turtle，所以这个文件的名字就不要用 turtle.py。这点很重要，后面学习到新内容时也一样，至于原因，等学到模块就会明白了。你也可以现在穿越到模块看一下。练习时如果实在没有好名字，你也可以命名为 turtle01.py。

“保存到文件”方法最直接的就是用记事本。不过，Python 环境提供了一个更方便开发 Python 程序的基本 IDE(集成开发环境)，叫作 IDLE。你可以在 Python 的安装目录中找到，也可以直接搜索运行 IDLE(见图 1-4)启动 PythonShell(见图 1-5)，PythonShell 是 IDLE 提供的一个交互界面，打开之后所处的环境跟在命令行下执行了 Python 之后所进入的 Python 环境一样，你可以直接执行 Python 指令但不能执行系统命令，要跟 PowerShell 区分开。在这里你还可以直接开始编辑一个文件，并且运行程序。这个过程比记事本方便很多，按图索骥吧！



图 1-4 搜索运行 IDLE

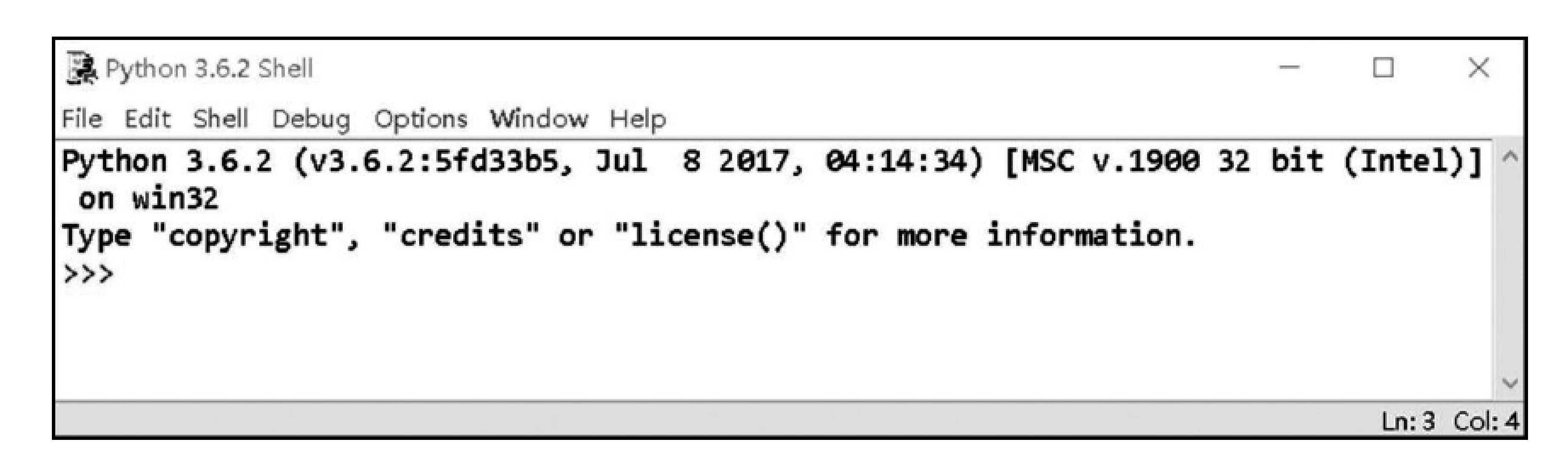


图 1-5 Python 3.6.2 Shell

除了交互界面，还可以在这里编辑一个文件，如图 1-6 所示。

然后直接编辑五角星程序（见图 1-7），最后保存下来（见图 1-8 和图 1-9）。

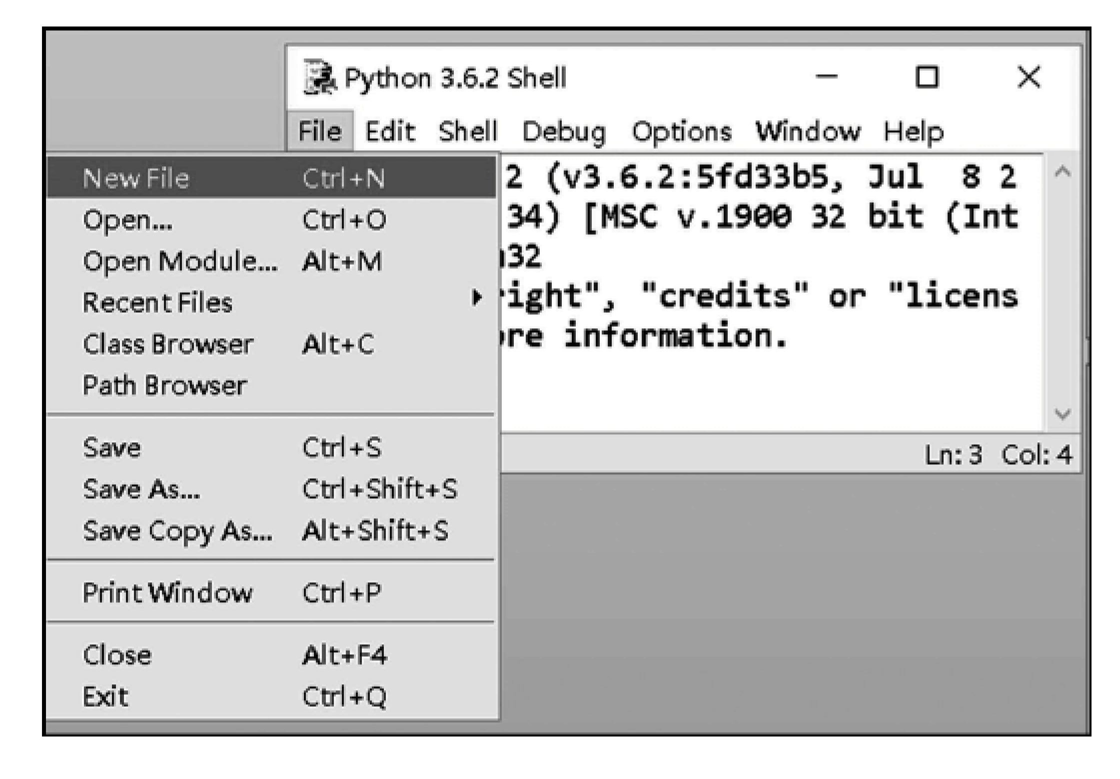


图 1-6　打开一个新文件

```
import turtle
turtle.forward(200)
turtle.right(144)
turtle.forward(200)
turtle.right(144)
turtle.forward(200)
turtle.right(144)
turtle.forward(200)
turtle.right(144)
turtle.forward(200)
turtle.right(144)
```

图 1-7　编辑五角星程序

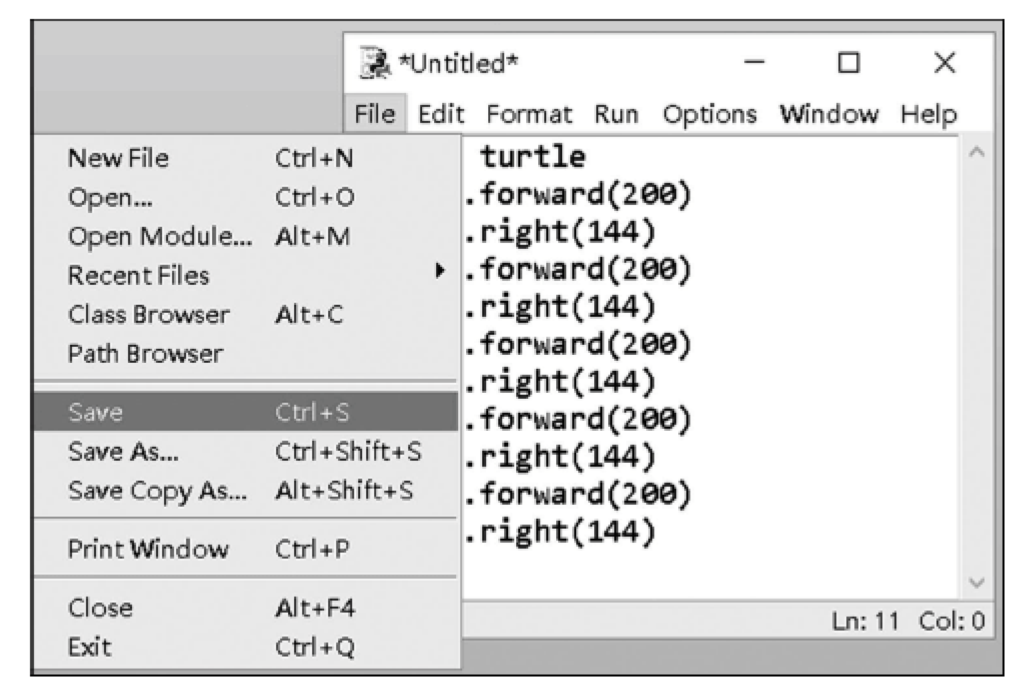




图 1-8　保存文件 1



图 1-9　保存文件 2

现在这个保存了指令集合的文件 wujiaoxing. py 就是你的第一个 Python 程序了，按 F5 键或者选择菜单命令 Run→Run Module 就可以让程序运行了（见图 1-10）。

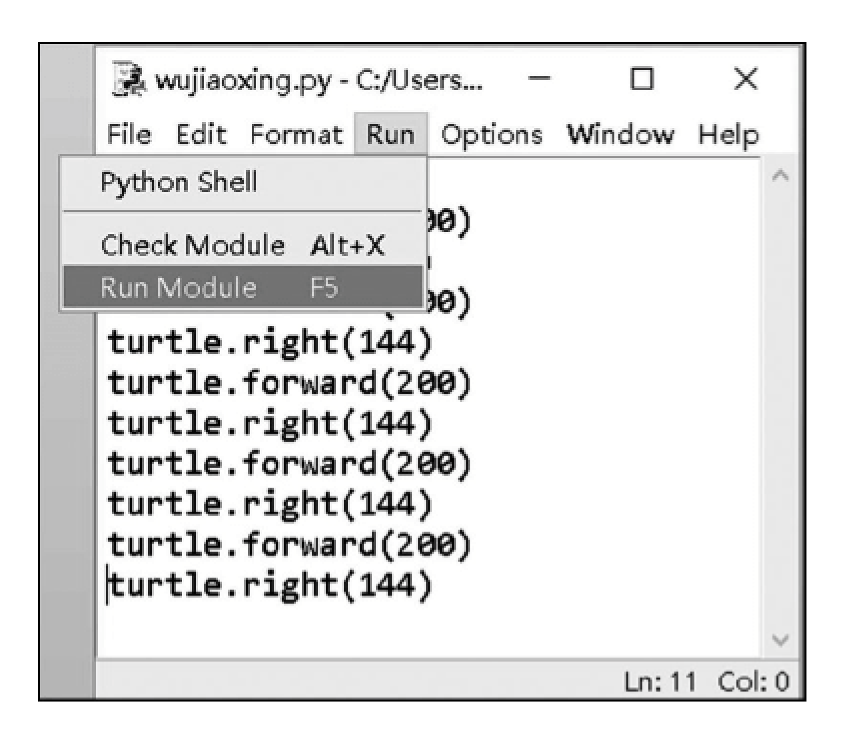


图 1-10　Run

最后，你会发现，五角星画完就消失了，这是因为程序所有指令执行完毕了，可以在程序最后加上一条 input()指令，再看看效果吧。input()的作用是等待用户输入数据，用户没有输入信息并回车之前会处于等待状态。



好了，这就是第一个程序的诞生和运行过程，是不是并不复杂？

1.2 熟悉开发环境，提高编程效率

前面我们利用 IDLE 运行 Python 程序，非常方便。另外，还有一种在命令行模式下通过 Python 命令直接执行的方式，在系统的命令行模式（比如 PowerShell）执行（见图 1-11）。



提高编程效率

```
PS C:\Users\milo>python C:/Users/milo/pycode/wujiaoxing.py
```

这里有一些前提，需解释一下：

（1）不建议这种模式执行。这种模式执行比较便利的前提是习惯在命令行模式工作的程序员，比如 linux、Mac 用户。

（2）这里的 PS C:\Users\milo>代表在系统命令行模式下，当前所在位置是 C 盘下 Users 目录下的 milo 目录中，python 是系统命令，这条命令不要进入到 Python 交互环境中执行。

（3）python 空格后面是要执行的程序文件，这里涉及文件路径问题，比如之前的五角星程序保存在 C:/Users/milo/pycode 路径下，所以这个命令是直接执行指定目录下的 wujiaoxing.py，这看起来有点麻烦。所以，你有两个选择：

① 所有程序都保存在打开命令行模式时的默认路径下，比如这里的默认路径就是 C:/Users/milo。

② 自己创建一个文件夹保存程序，执行之前切换到你创建的目录，比如我创建的是 pycode。

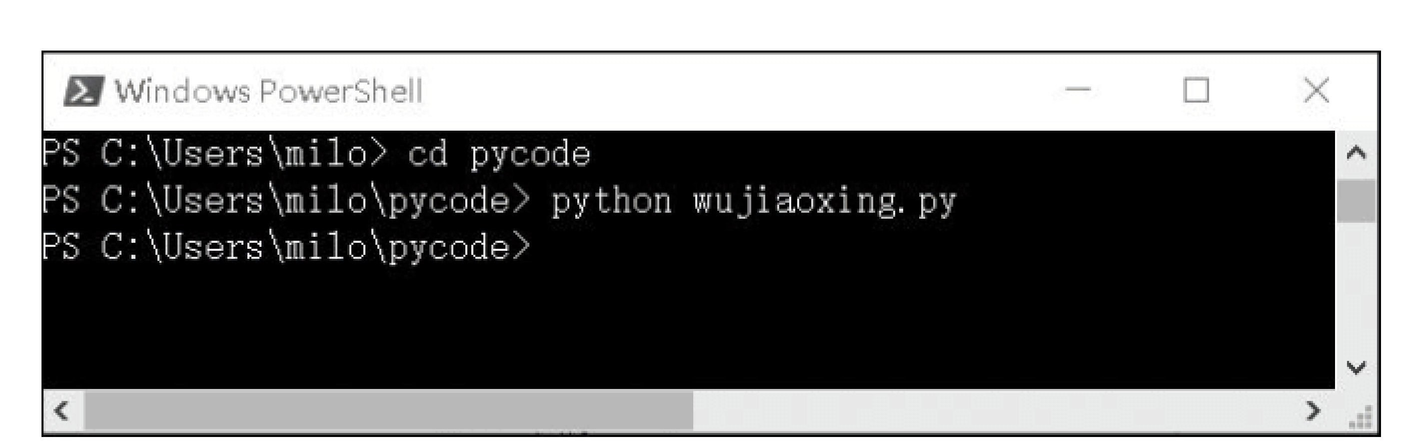


图 1-11 切换目录并执行程序

图 1-11 中 cd 是系统用来切换目录的命令，这条命令的意思是切换到当前目录下的 pycode 目录中，前提是 pycode 已存在。你可以通过 dir 命令查看当前目录下的文件列表。python 后面直接跟文件名，没有路径的意思，是直接执行当前目录下的程序。

在跨平台开发过程中，掌握命令行模式是很有必要的，这不仅会让你在外人眼里看起来像个高手，实际上也会让你如虎添翼。

1.3 Python 开发工具

除了 Python 自带的 IDLE 和记事本以外，还有很多流行的开发工具，在实际工作中可以选择一个适合自己的。

Vim、Emacs：老牌编辑器，类似 Winows 的记事本，是神一样的存在，Vim 更是被称为编辑器之神。如果不是 Windows 用户，强烈推荐使用；Emacs 则被称为神的编辑器，一般人用不了，太烦琐。

notepad++、editplus：Windows 下的老牌编辑工具，不过，此老非彼老，地位比 Vim 差很多，但作为比较轻量化的工具还是很好用的。

Pydev、Pycharm、Spyder：都是比较专业的集成环境。Pycharm 比较好用，分商业版和共享版，推荐使用，不用 Vim 就推荐 Pytharm，其他开发工具可根据习惯和实际工作环境决定是否使用。

另外，还有诸如做科学计算的集成环境 python(x,y)等，包含了非常全面、做科学计算会用到的包，还包含了 Spyder。

在这里我仅做一点说明，初学时不必刻意使用这些 IDLE，Python 自带的 IDLE (PythonShell)就足够用了，而且我也建议使用系统自带的、最纯粹的方式去学习。很多教编程的老师一上来搭完环境，就开始用一些很先进的工具，看起来好像很专业，实际对新手了解基本工作原理没有帮助，况且这些工具，有些做了高度的包装和功能集成，新手经常会无从下手。但是如果你只会用自带的 IDLE，作为实战开发来讲，效率是不够的。建议在学完全部语法之后开始使用这些 IDLE，并且找到适合自己的。

1.4 第三方模块和工具管理

Python 除了自带的模块之外，还有很多第三方模块和工具，这些第三方模块功能各异，有的功能只需要一个模块就可以实现，比如 turtle，有的是几个库协同工作完成一个任务，比如机器学习可能需要 scikit-learn、numpy、pandas、scipy 等。

当需要第三方库时，有两种选择：一种是初学阶段推荐采用的方式，就是需要什么就一个一个地安装；另一种是直接安装诸如 Anaconda 这样的包管理器和环境管理器。Anaconda 附带了一大批常用的数据科学包，包括 conda、Python 和 150 多个科学包及其依赖项，因此可以立即开始处理数据。Anaconda 是在 conda(一个包管理器和环境管理器)基础上发展出来的。conda 可以帮助你为不同的项目建立不同的运行环境，比如在项目 A 中用了 Python 2，而在项目 B 使用了 Python 3(同时安装两个 Python 版本可能会造成许多混乱和错误)，这时候 conda 就可以帮你解决问题，不过这本书不会过多介绍 Anaconda，你也不要着急现在了解，学到一定程度时，若有需要，再自行学习即可。

现在我们要解决一个实际问题，你已经知道在命令行模式中不能执行 Python 指令，在 Python 中也不能执行系统命令，那如果我们在 Python 环境下测试指令时又需要执行一些系统命令，该怎么办呢？退出 Python，执行完系统命令之后再进去？这样是可以，但是却很麻烦，实际上如果开发的程序跟系统文件有关系，这样的操作会很频繁。

所以，我们来认识一个新的工具 Ipython，并且通过安装 Ipython，了解 Python 怎样管理第三方库。

Ipython 是增强的 PythonShell，不仅可以执行 Python 指令，还支持执行常用系统命令。

Ipython 是一个独立的工具，相当于在你的 Python 之上加了一层增强外壳，所以需要先安装 Python 再安装 Ipython。Ipython 程序源代码和其他 Python 模块一样，都托管在

PyPI 上(见图 1-12),目前,权威的 PyPI(Python Package Index) 地址是 https://pypi.org/,这个 PyPI 就像个大仓库一样,pypi.python.org 是重定向到官网的。

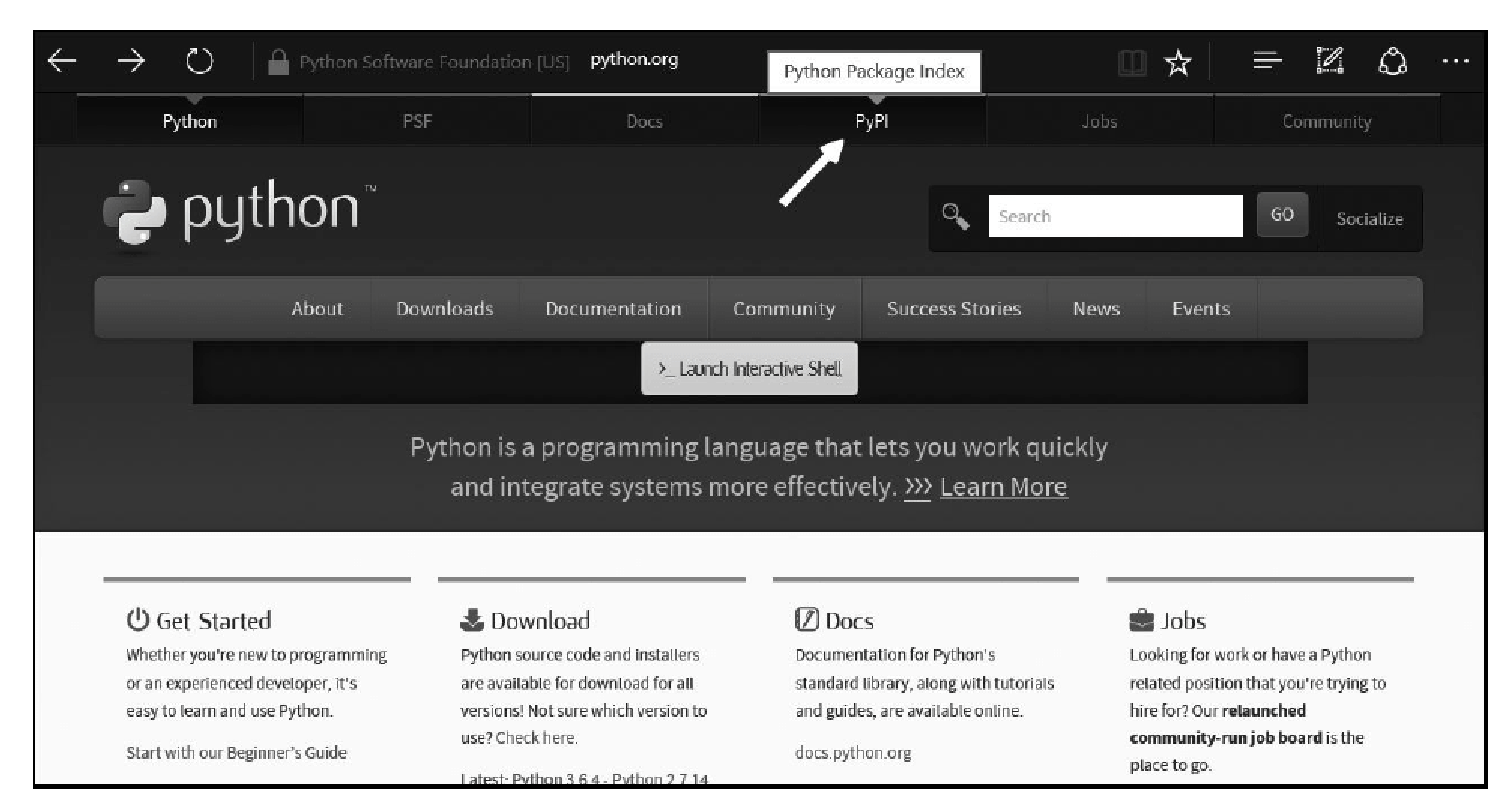


图 1-12　python.org 网站

这些在安装 Python 时不具备的库,称为第三方库,当然,你并不需要打开网页下载,可以通过 pip 命令很轻松地管理 PyPI 上的这些库(我们在安装 Python 时的可定制步骤已经见过 pip,如果没有 pip 命令,那么快想想原因吧)。后面我们还会用 pip 安装其他库,这里先用 Ipython 做一个第三方库管理的练习(见图 1-13)。

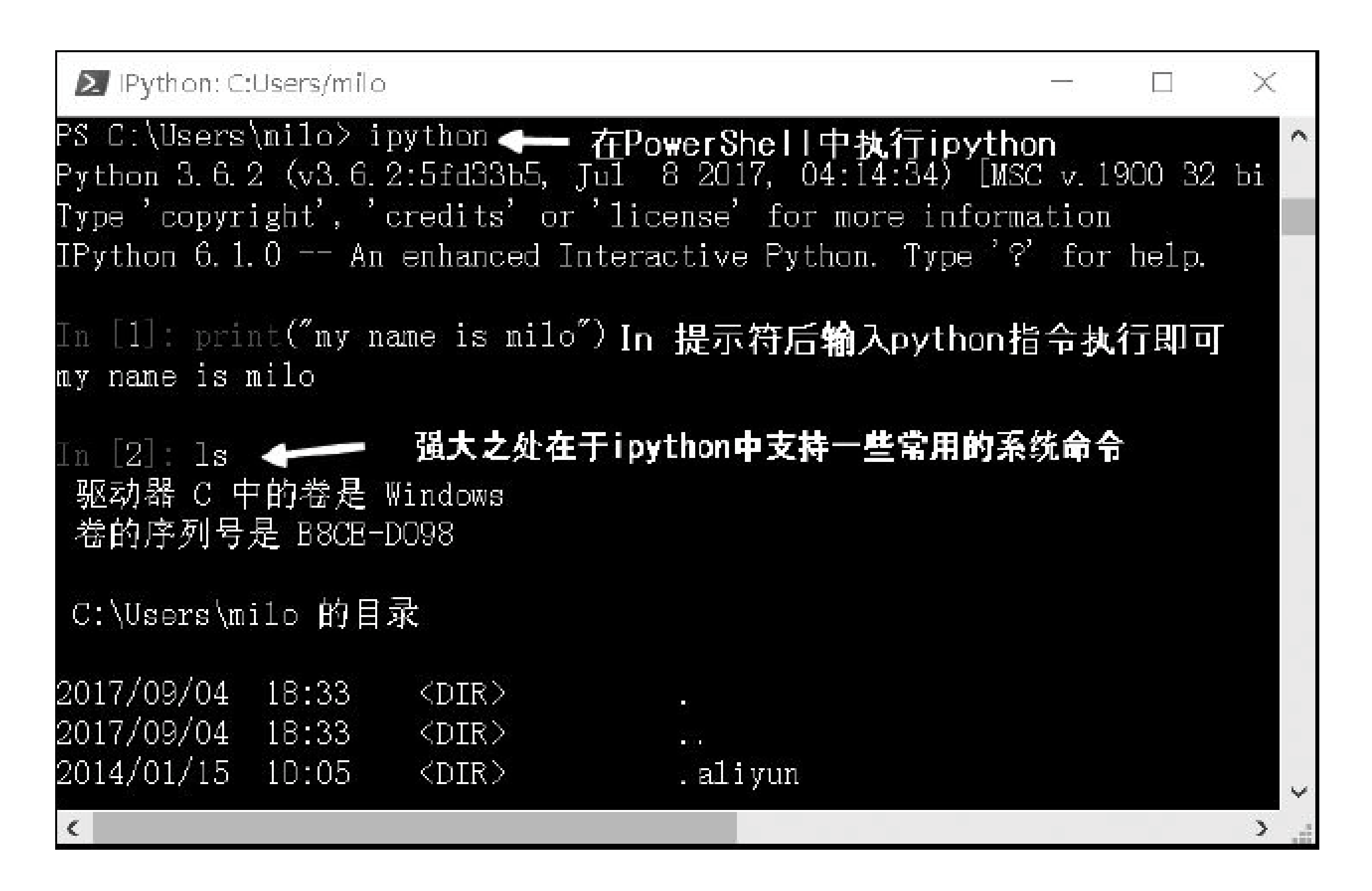


图 1-13　Ipython 环境

(1) 安装(注意大小写)

```
PS C:\Users\milo>pip install Ipython
```

(2) 查询

```
PS C:\Users\milo>pip show Ipython
```

（3）卸载

```
PS C:\Users\milo>pip uninstall Ipython
```

（4）升级，如果未安装则直接安装

```
PS C:\Users\milo>pip uninstall -U Ipython
```

安装后我们来试运行一下。

Ipython 还有很多强大的功能，在初期作为辅助工具，暂时不做过多介绍。

1.5 像程序员一样写代码



Python 程序员怎样写程序



Python 程序员怎样写程序：缩进

前面说过，程序不是写给自己看的，通常也不是一个人写的，所以，养成良好的编码习惯至关重要。有很多具有多年编程经验的程序员，工作态度不端正，编写的程序功能上差不多就行了。这类程序员设计的程序本身并不精巧，结构混乱，甚至只是盲目地堆砌代码，仅仅以实现功能为准，连注释也写得稀里糊涂。你可能认为这是个别现象，但其实，我说的这些并不是偶然现象。所以，如果有那么多不认真的程序员，那至少自己做一股清流吧！保持正确姿势起飞，才能越飞越高。

现在，画五角星的方法有了，写代码的工具有了，代码也写了，程序就已经是能运行的程序了。但是，能运行的程序不一定是好程序，所以，在写更多的代码之前，我们先把程序的组成和写代码基本的规则了解一下。

什么样的代码是好代码？这个定义起来似乎有说不完的话题，不过如果你清楚程序的组成部分，能做到遵循语言的编码规范，那么，你写出来的代码至少看上去不会太糟糕。

程序的组成按功能分，有注释、模块、表达式和语句、空白。

对于 Python 的编码规范，官方有专门的文档 PEP 8，可以通过 python. org 找到，如图 1-14 所示。

对初学者来说，肯定会觉得文档内容太多，比如涵盖了空行、空格、注释等，而且很多内容还没有学到，不知道是什么意思，所以不必现在就搞清楚所有编码规范，你只要在脑子里记住有一个编码规范就好，本章将结合程序的组成部分和编码规范做一些说明。

1.5.1 注释

注释在程序运行过程中是可有可无的，它是附加在程序上的说明或者简单的标识。但是，能否写好注释或者正确地做注释是一个优秀程序员的必修课，因为在实际工作中，涉及团队协作的项目，或者更改别人的代码，都会涉及代码的调用和调试，能够在最短时间内理解程序的作用，对提高工作效率是非常有帮助的。

Python 中的注释有单行注释（行注释和块注释）和多行注释（文档字符串 docstring）。

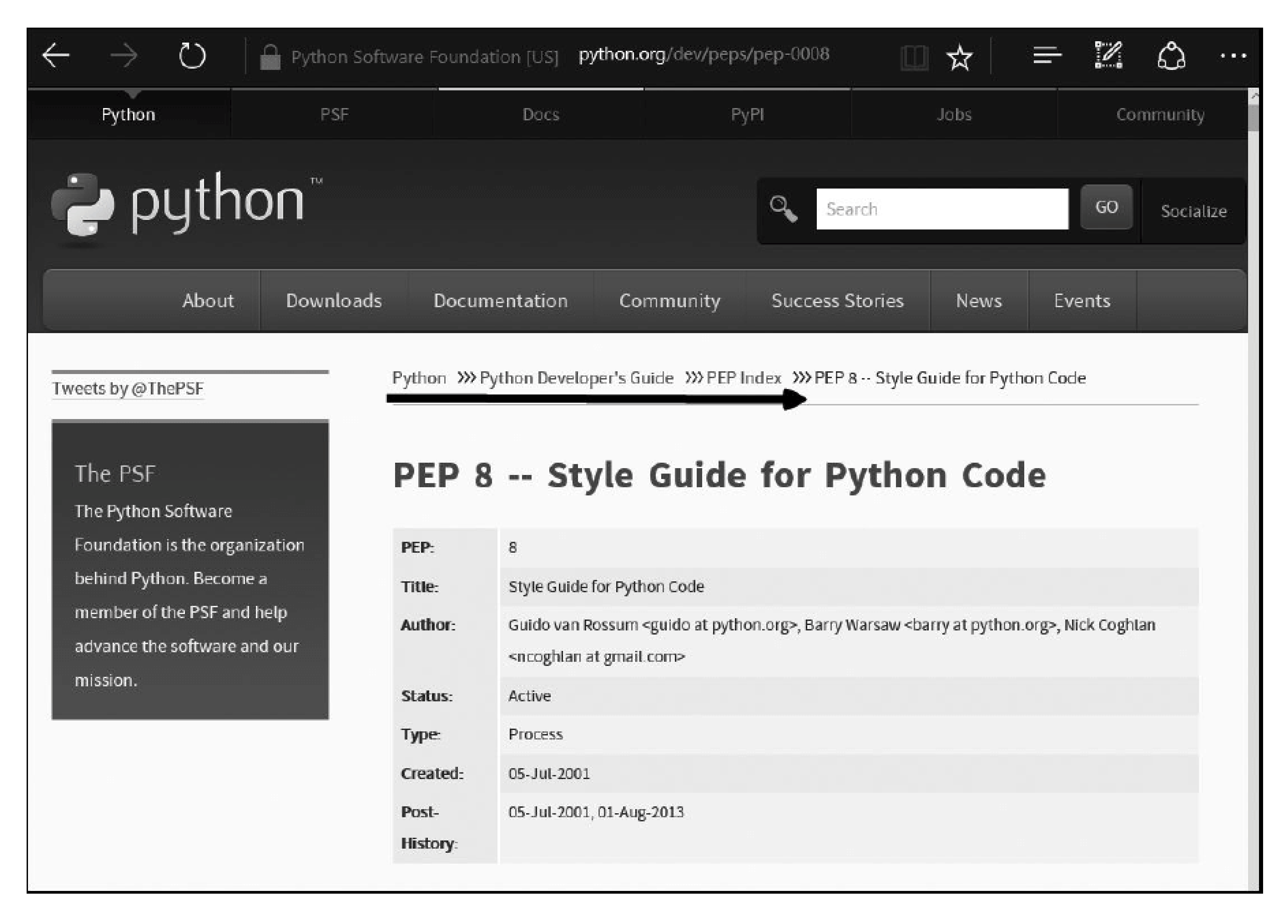


图 1-14　PEP 8

还有一些特殊形式的注释，比如关于中文的注释。下面我们分开来讲。

1. 单行注释（行注释和块注释）

Python 中使用#作为单行注释符，在程序执行过程中，#右侧的代码或者内容是不会被执行的。最需要写注释的是代码中那些技巧性的部分，如果在下次代码审查时必须解释一下，那么你应该现在就给它写注释。对于复杂的操作，应该在其操作开始前写上若干行注释。对于不是一目了然的代码，应在其行尾添加注释，为了提高可读性，注释应该至少离开代码 2 个空格。例如：

```
#We use a weighted dictionary search to find out where i is in
#the array.  We extrapolate position based on the largest num
#in the array and the array size and then do binary search to
#get the exact number.

if i & (i-1) == 0:        #true if i is a power of 2
```

另一方面，绝不要描述代码(假设阅读代码的人比你更懂 Python，他只是不知道你的代码要做什么)。

```
#BAD COMMENT: Now go through the b array and make sure whenever i occurs
#the next element is i+1
```

2. 多行注释（文档字符串 doctring）

如果需要注释的内容是一个段落，就需要进行多行注释。Python 有一种独一无二的注

释方式：使用文档字符串，也叫 docstring。文档字符串是包、模块、类或函数里的第一个语句。这些字符串可以通过对象的__doc__属性被自动提取（对象概念属于后面的面向对象部分），我们对文档字符串的惯例是使用三重双引号"""（PEP-257）。一个文档字符串应该这样组织：首先是一行以句号、问号或惊叹号结尾的概述（或者该文档字符串单纯只有一行）；接着是一个空行；然后是文档字符串剩下的部分，它应该与文档字符串第一行的第一个引号对齐。例如：

```
class SampleClass(object):
    """Summary of class here.

    Longer class information...
    Longer class information...

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

如果单双引号要细分，一般类文档、函数文档、字符串之类的用双引号，变量用单引号。

3. Shebang

在计算机科学中，Shebang（也称为 Hashbang）是一个由＃号和叹号构成的字符串行（＃!），其出现在文本文件第一行的前两个字符。

如果看源代码，你会发现在 Linux/UNIX 系统下的 Python 程序会经常在第一行就看到＃!/usr/bin/env python 这样一条注释，它的作用是告诉 Linux/UNIX 去找到 Python 的解释器，就好像给系统指路一样。在脚本被直接运行时会起到作用（Windows 下没用），否则，系统会因为找不到 Python 解释器而无法运行脚本。需要注意的是：

（1）必须是文件的第一行。

（2）必须以＃!开头。

大部分.py 文件不必以＃!作为文件的开始。根据 PEP-394，程序的 main 文件应该以＃!/usr/bin/python2 或者 ＃!/usr/bin/python3 开始。在文件中存在 Shebang 的情况下，类 UNIX 操作系统的程序载入器会分析 Shebang 后的内容，将这些内容作为解释器指令，并调用该指令，并将载有 Shebang 的文件路径作为该解释器的参数。例如，以指令＃!/bin/sh 开头的文件在执行时会实际调用/bin/sh 程序。

＃!用于帮助内核找到 Python 解释器，但是在导入模块时会被忽略。因此只有被直接执行的文件中才有必要加入＃!。

4. 中文问题

如果程序中使用了中文,不论是执行代码,还是注释内容。都要加上中文编码的特殊注释,例如:

```
#-*-coding: utf-8-*-
```

看到#,你会想到这是一个单行注释,关键在于后面的 utf-8,这里涉及关于中文编码的问题。

Python 解释器在加载 .py 文件中的代码时,会对内容进行编码(默认 ASCII)。

Python 2 中,文件的默认编码为 ASCII,在文件中含有中文时会报错,因此要在文件开头写上:

```
#-*-coding: utf-8-*-
```

指定编码类型为 UTF-8。

Python 3 中,文件的默认编码为 UTF-8,已经不存在上述问题了。

小知识:

ASCII(American Standard Code for Information Interchange,美国标准信息交换代码)是基于拉丁字母的一套计算机编码系统,主要用于显示现代英语和其他西欧语言,其最多只能用 8 位来表示(一个字节),即 $2^8=256$,所以,ASCII 码最多只能表示 256 个符号。

显然 ASCII 码无法将世界上的各种文字和符号全部表示,因此就需要一种可以代表所有字符和符号的编码,即 Unicode。

Unicode(万国码)是为了解决传统的字符编码方案的局限而产生的,它为每种语言中的每个字符设定了统一且唯一的二进制编码,规定所有的字符和符号最少由 16 位来表示(2 个字节),即:$2^{16}=65536$。

UTF-8 是对 Unicode 编码的压缩和优化,它不再局限于最少使用 2 个字节,而是将所有字符和符号进行分类:ASCII 码中的内容用 1 个字节保存、欧洲的字符用 2 个字节保存,东亚的字符用 3 个字节保存……

最后,对注释的使用做一个总结说明。

(1) 为了避免不必要的麻烦,必要时可以直接在文件的开头加上:

```
#!/usr/bin/env python
#-*-coding: utf-8-*
```

(2) 每一个 Python 文件的开头,第一条的两行代码之后,经常要写上关于这个模块,也就是这个 Python 文件实现的功能、注意事项、可能会发生的错误,总之注释要让使用者对程序有个直观认识,例如:

```
"""
requests.cookies
~~~~~~~~~~~~~~~~
```

```
Compatibility code to be able to use 'cookielib.CookieJar' with requests.
requests.utils imports from here, so be careful with imports.
"""
```

做到不看代码，就已经知道接下来的是什么了。

(3) 每一个类下面加上关于这个类的说明以及用法，这样使用程序的人不需要知道程序内部构造，就可以使用了，例如：

```
class HTTPAdapter(BaseAdapter):
    """The built-in HTTP Adapter for urllib3.
    Provides a general-case interface for Requests sessions to contact HTTP and
    HTTPS urls by implementing the Transport Adapter interface. This class will
    usually be created by the :class:'Session ' class under the covers.
    :param pool_connections: The number of urllib3 connection pools to cache.
    :param pool_maxsize: The maximum number of connections to save in the pool.
    :param max_retries: The maximum number of retries each connection
    should attempt. Note, this applies only to failed DNS lookups, socket
    connections and connection timeouts, never to requests where data has
    made it to the server. By default, Requests does not retry failed
    connections. If you need granular control over the conditions under
    which we retry a request, import urllib3's ''Retry'' class and pass
    that instead.
    :param pool_block: Whether the connection pool should block for connections.
    Usage::
    >>>import requests
    >>>s =requests.Session()
    >>>a =requests.adapters.HTTPAdapter(max_retries=3)
    >>>s.mount('http://', a)
    """
```

第一行定义了一个类，类内部的三引号注释对这个类“是干什么的”“经常在什么情况下使用”“如何使用”都做了很详细的说明。

(4) 每一个函数下面务必加上多行注释，当然，如果函数注释只有一行或者两行，可以使用单行注释，也可以使用多行注释，这里与类函数说明相当，注释中往往包含使用说明及注意事项。例如：

```
def __setstate__(self, state):
    #Can't handle by adding 'proxy_manager' to self.__attrs__ because
    #self.poolmanager uses a lambda function, which isn't pickleable.

def has_capacity(self):
    """Does the engine have capacity to handle more spiders"""
    return not bool(self.slot)
```

(5) 在必要的地方加上单行注释，例如：

① 你不太理解的代码。



② 别人可能不理解的代码。

③ 提醒自己或者别人注意的代码和重要的代码。

总之，注释很重要，但不是越多越好，那些能够一眼就看懂的代码，就无须画蛇添足了。

1.5.2 模块导入

模块是 Python 的命令集（通常就是 Python 文件），可以在一个脚本中导入另一个模块，也可以把模块导入到 PythonShell 的交互模式下。导入方式如下：

```
>>>import module
```

模块是 Python 的重要组成部分，也是 Python 之所以强大的原因，安装 Python 时，其中自带的模块有二三百个，但对于有些工作来讲，这还是不够的。所以，我们还可以安装第三方模块，而且既然模块就是 Python 脚本，那么，你完全可以自己编写模块完成私人订制。至于结构，从 turtle 绘图的例子，大家应该也发现了，如果要想使用模块的功能，就要先导入。从能用的角度来说，现用现导就行；从结构来说，导入模块的语句通常都在最前面，这样比较直观，一眼就知道当前程序使用了哪些模块。

1.5.3 表达式和语句

Python 中的代码可以分成两类：表达式和语句。

（1）表达式：值和运算符的组合，会产生新值，这一点跟数学表达式的定义是一致的。比如，在交互模式输入 1 + 1 会显示 2，这就是说表达式 1 + 1 显示返回值 2。

（2）语句：执行任务的指令，没有返回值。比如，x = 1 + 1 是一个语句，其中的 1 + 1 是表达式。

```
>>>x = 2        #语句
>>>x + 3        #表达式
5               #返回值
>>>y = x + 3    #语句，等号右侧的 x + 3 是表达式
>>>y            #语句
5               #这个 5 不是返回值，程序中这样执行看不到结果，交互会话中能看到
```

1.5.4 合理利用空白

艺术讲究留白，写程序也一样，代码中合理地使用空格、空行、缩进等，可以大大地提高代码的可读性。但是，如果滥用，虽然运行可能没问题，但对于阅读代码的人来说却是灾难。本书中有些例子为了考虑版面可能不会完全遵循这个原则。但是，你在写代码时要养成好习惯，多些空白并不会造成太多浪费。

1. 空行

顶级定义之间空两行，比如函数或者类定义。方法定义、类定义与第一个方法之间都应该空一行；函数或方法中，某些地方若合适就空一行。主要目的还是为了阅读方便。

2. 空格

空格是在代码中使用最多的，可能也是最乱的，有时，跟个人习惯有很大关系。建议大家按照标准排版规范使用标点两边的空格。

(1) 不要在逗号、分号、冒号前面加空格，在它们后面加，如果是行尾，则不加；括号内不要有空格。

```
正确: spam(ham[1], {eggs: 2}, [])
错误:spam( ham[ 1 ], { eggs: 2 }, [ ] )
```

(2) 在二元操作符两边都加上一个空格，比如，赋值(=)、比较(==、<、>、!=、<>、<=、>=、in、not in、is、is not)，布尔(and、or、not)。

```
正确:x == 1
错误:x<1
```

(3) 当=用于指示关键字参数或默认参数值时，不要在其两侧使用空格。

```
正确:def complex(real, imag=0.0): return magic(r=real, i=imag)
错误:def complex(real, imag = 0.0): return magic(r = real, i = imag)
```

(4) 不要用空格来垂直对齐多行间的标记，看起来很整齐，但是这会成为维护的负担(适用于:、#、=等)。

```
正确:
    foo = 1000       #等号不需要对齐
    long_name = 2    #注释不需要对齐

    dictionary = {
        "foo": 1,
        "long_name": 2,
        }
错误:
    foo       = 1000  #等号不需要对齐
    long_name = 2     #注释不需要对齐

    dictionary = {
        "foo"      : 1,
        "long_name": 2,
        }
```

3. 缩进

Python 中的缩进除了可以用来对齐相同特征的元素，使代码更具可读性之外，更重要的是，可以用来分组。对于需要组合在一起的语句或表达式，Python 采用相同空格的缩进进行区分，比如函数和它的代码块。

在 Python 程序员的世界里，使用 4 个空格缩进代码很重要。因为不这样做，可能会让你在同行中看起来很不专业。当然，后面实验时你会发现几个空格都行，用 Tab 键也行，但是在你明确知道用 Tab 键的意义之前不要用 Tab 键(在有些 IDLE 里面可以设置 Tab 键为 4 个空格，这种情况下 Tab 键等同于 4 个空格)，更不能空格和 Tab 键混合使用。

下面看一下对齐的使用形式。

```
正确：
#与起始变量对齐
foo = long_function_name (var_one, var_two,
                          var_three, var_four)

#字典中与起始值对齐
foo = {
    long_dictionary_key: value1 +
                         value2,
    …
}

#4 个空格缩进，第一行不需要
foo = long_function_name(
    var_one, var_two, var_three,
    var_four)

#字典中 4 个空格缩进
foo = {
    long_dictionary_key:
        long_dictionary_value,
    …
}
错误：
#第一行有空格是禁止的
  foo = long_function_name(var_one, var_two,
   var_three, var_four)

#2 个空格是禁止的
  foo = long_function_name(
  var_one, var_two, var_three,
  var_four)

#字典中没有处理缩进
  foo = {
   long_dictionary_key:
    long_dictionary_value,
    …
  }
```

重点提示

在这一章中,你要掌握的内容:

(1) 明确 import 的作用。

(2) 熟悉 Python 的交互模式并在交互模式下执行 Python 指令。

(3) 熟悉 IDLE 并编写执行第一个 Python 程序。

(4) 会在命令行模式执行 Python 程序。

(5) 能通过 pip 管理第三方模块(库)。

(6) 明确编码规范。

动动手

(1) 通过 pip 安装 Ipython。

(2) 分别在 Python 和 Ipython 中用 turtle 模块画一个五角星。

(3) 通过 IDLE 编写并保存五角星的脚本到上一章创建的 My Python Code 文件夹,用 Run 命令和 F5 键各执行一次。

(4) 在命令行模式下找到五角星脚本,直接在命令行执行。



挑战自己

现在,你已经可以用 turtle 画五角星了,从现在开始到接下来的学习过程中你可以试着完成一项新手任务挑战——画美国国旗。这个挑战的目的在于培养独立解决未知问题的能力,请相信,这将会是你成为程序员之后必备的能力。如果在这个过程中你觉得有难度,可以在后续的学习过程中,通过新的知识来提高自己的能力,解决碰到的问题,完成这个挑战会使你对语法基础的掌握有很大帮助。

美国国旗于 1777 年 7 月 14 日正式被规定并且开始使用。第一面美国国旗(见图 1-15) 13 个星,13 个条。

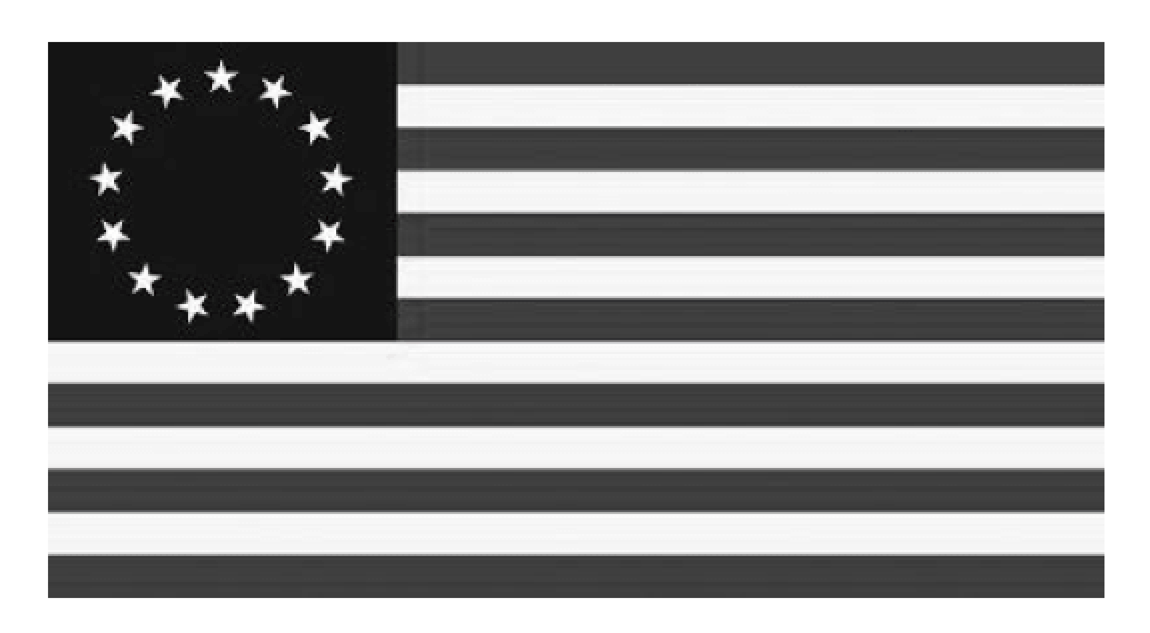


图 1-15 第一面美国国旗

随着行政版图的变化,美国国旗也在不断变化,1818 年的第三面美国国旗(见图 1-16),就变成了 20 个星星,13 个横条。这个星星排列比较规则,可以先画这个。

建议:使用至少 4 个函数。美国国旗中正多边形都作为函数处理。现在画不出来,可以等到学完函数再画。

提示：画行，画圆，这里的圆可以当作多边形处理。

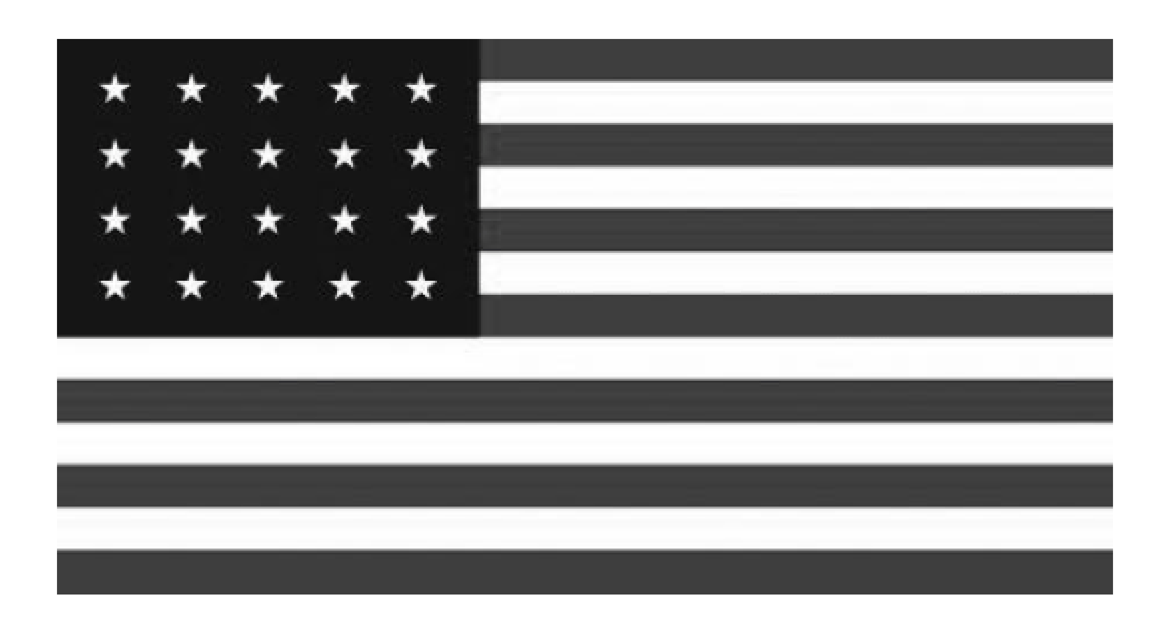


图 1-16 第三面美国国旗

第 2 章 程序员基础

相信现在你对编程已经有兴趣了。Python 还有很多模块，而且大部分像 turtle 模块一样，看懂说明书就能用。但这并不能解决所有问题，况且没有扎实的基础，根本看不懂模块的源代码，程序员大部分工作都是在编码写程序，而程序就是由最基本的语句按照一定的语法组织起来的。所以不知道语句怎么写，不知道语法规则，有再多模块也是没用的。

2.1 程序开发全局观

初学编程的人经常会被一些琐碎的知识点牵绊，也可能会纠结于某一个功能或某几行代码。比如，我们打算开发一个游戏需要做些什么事呢？先闭上眼睛，在脑子里想一想，第一个想法是代码怎么编写吗？应该不是，如果不出意外，你的想法大概是“开发一个游戏”“什么样的游戏”“怎么玩”“游戏里面都能干什么”诸如此类，应该不会马上就想到要怎么写代码，更不会有人马上打开计算机开敲键盘。所以，程序员在实际工作中更需要有全局观，从整体考虑程序的实现过程。

一个程序从无到有，简单说就是三步：需求分析、程序设计和编码测试。下面我们就在这些关键点，用一个虚拟项目《英雄无敌》来做说明和演示。

为了更直观地感受所学习的知识在程序设计中的体现，在本书适当位置，会借助《英雄无敌》这个虚拟项目，把需要用到的知识点更直观、更生动地引出来，实现从 0 到 1 的过程，然后通过延伸问题的引导，在迭代开发的过程中，完成属于自己的《英雄无敌》。



《英雄无敌》迭代开发游戏：需求分析、实现

按项目开发的流程，首先进行需求分析。这是开发人员经过深入细致的调研和分析，准确理解用户和项目的功能、性能、可靠性等具体要求，将用户的需求表述转化为完整的需求定义，从而确定系统必须做什么的过程。比如针对《英雄无敌》，作为用户会想到一些初步的零散需求：

(1) 注册、登录、验证。

(2) 给角色起个名字，初始化英雄。

(3) 游戏的开场前奏。

(4) 英雄人物满血出场。

(5) 有地图，可以在地图上走。

(6) 在地图上行走时，发生随机事件。

通常这样的需求很多，项目越大，参与的人越多，需求就越复杂，而且通常还会变来变去，程序员的工作就是把这些需求变成代码。现在你自己是用户兼产品经理兼程序员，需求可以根据你掌握的编程能力进行适当调整。不过，建议从需求出发，想到什么功能，就想办法去实现它。

对需求进行深入细致分析后，先不要马上开始写代码，而是要构思程序结构以及实现的功能，最后才落到编码上，然后进入编码测试阶段，实现功能之后再迭代优化。

针对上述需求，并提炼简化，列出首先要实现的功能：

(1) 游戏开局，玩家输入角色名字。

(2) 判断用户名字是否为空，若为空，则英雄名字初始化为player01。

(3) 初始化英雄属性，包括名字、血值100。

(4) 交互提示游戏流程信息。

(5) 设计一个线形地图，玩家可以在地图上移动。



《英雄无敌》迭代开发游戏：coding

有了精确的功能列表(作为菜鸟的第一个需求就是能提炼出核心问题)就很容易写出代码了，下面就是简陋的《英雄无敌》0.1版。

```
'''
#Heroes-0.1.py
Heroes beta-0.1
milo
'''

hp = 100

print('welcome Heroes world!')
print("|the world is like this #####,'a'for left,'d' for right |")

name = input('input your name:')

if not name:
    name = 'player01'

usermsg = [name,hp]

print("your hero's name is:",usermsg[0],'Hp is :',usermsg[1])
print("and now you are here: * ####|'a'for left,'d' for right |")

userinput = input()

if userinput == 'd':
print("your are here # * ###")

if userinput == 'a':
    print("your are here * ####")
```

怎么做到有了几条需求就写出这么多东西呢？别着急，代码看上去很多，其实核心功

能并不复杂，都是很直白的功能，仔细看看代码里面的单词，就能想到是什么样子，运行起来是这样的。

```
>>>
========== RESTART: C:\Users\milo\pycode\Heroes-0.1.py ========
welcome Heroes world!
|the world is like this #####,'a'for left,'d' for right |
input your name:milo
your hero's name is: milo Hp is : 100
and now you are here: * ####|'a'for left,'d' for right |
d
your are here # * ###
>>>
```

这段代码用到的知识有变量、结构化数据、输入/输出和判断。这些都是基础语法，掌握的语法越多，能实现的功能就越高级，现在这还是个很烂的程序，如果你试着运行会发现，这段程序只能顺序执行到底，比如英雄选择了一次行进方向，游戏就结束了，没有可玩性。随着学习的深入，可以考虑新学到的知识点是否能解决你已经设计好的功能。等学完流程控制，就可以写出一个具有可玩性的小游戏了。

养成从全局考虑问题的习惯，不断完善自己的知识体系，这也是一个程序员的自我修养。在实际工作中总是会有新的知识或者不知道的技术，但是，只要从需求出发，其他的无非就是查资料、看文档，现学现卖也好，快速消化新知识也好。总之，只要有想法，技术问题就不是问题。

2.2 数据的标签：变量

变量一词来源于数学，是编程语言中最常见的组成部分，用来表示计算结果或表示抽象概念。变量可以用来表示一个值、一组数据、一个文件，甚至另一个程序，对于初学者来说，基本上变量就是一个值，可能是一个数字，也可能是一串字符。由于变量能够把程序中准备使用的每一段数据都赋给一个简短或易于记忆的名字，这个名字就像数据的标签一样，通过名字就可以得到对应的数据，因此十分有用。变量的值可以由程序员直接赋值，也可以是用户输入的数据、特定运算的结果以及一个内涵丰富的对象等。简而言之，变量是用于跟踪几乎所有类型信息的简单工具。


变量

2.2.1 声明变量

声明变量也叫定义变量，声明变量也是变量赋值的过程，这个过程就是给变量起一个名字，然后给它一个数据，比如英雄的血值：

```
>>>hp = 100
```

这个指令的作用就是定义一个名字为 hp 的变量，通过等号把值整数 100 赋给它。

1. 变量的命名

一个好的程序，其中使用的名字一定是有助于增加程序可读性的。比如，当你看到 Pi 这个名字时就会想到圆周率；看到 password 就会想到密码，诸如此类。除了简洁明了之外，Python 中有自己的命名规则，下面的命名规则通用于变量名、函数名、类名等。

（1）名字由字母、数字、下划线“_”组成，数字不能作为首字符。

（2）名字长度不限，但是要考虑可读性。

（3）严格区分大小写。

（4）不要使用 Python 关键字（关键字是预先保留的标识符，每个关键字都有特殊的含义）。可以通过下面的指令获得 Python 关键字信息（不同版本可能会有区别）。

```
>>>import keyword
>>>keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def',
'del', 'elif', 'else','except', 'finally', 'for', 'from', 'global', 'if', 'import',
'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try',
'while', 'with', 'yield']
```

此外，在命名习惯上，大多数程序员更习惯于驼峰命名法，驼峰命名法比较形象，是复合词或短语的常用写法，单词之间没有空格和下划线。单词通过首字母大写进行区别，看起来就像驼峰一样，比如 ThisIsAnClass、myFunction。除第一个单词外，其他单词首字母大写叫小驼峰；把第一个单词的首字母也大写叫大驼峰。在 Python 之父 Guido 推荐的规范中，类命名时使用大驼峰，但是模块名应该用小写加下划线的方式，如 lower_with_under. py。尽管已经有很多现存的模块使用类似于 CapWords. py 这样的命名，但不鼓励这样做，因为如果模块名碰巧和类名一致，会造成不必要的困扰。由于编程语言众多，程序员转来转去，造成风格习惯不一，如果不能完全做到统一命名，至少在自己的项目团队中保持统一。

另外，在 Python 中以下划线“_”开头的变量具有特殊意义，在明确知道其意义前，最好不要以下划线“_”开头（面向对象章节有详细讲解）。Python 之父 Guido 推荐的变量命名规范如表 2-1 所示。

表 2-1　Guido 推荐的变量命名规范

类　　型	公有变量	私有变量
Modules	lower_with_under	_lower_with_under
Packages	lower_with_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected) or __lower_with_under (private)

续表

类　　型	公有变量	私有变量
Method Names	lower_with_under()	_lower_with_under() (protected) or __lower_with_under() (private)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

2. 赋值

Python 的赋值符号是等号“＝”，赋值的目的就是将名字和值关联起来，比如：

```
x = 1
y = x + 1
```

虽然使用的是等号，但是不同于数学的等号，在数学计算中通常左侧是运算，右侧是结果，而程序语言中的赋值操作是先对右侧表达式进行计算，再将结果赋值给左侧变量名。事实上，赋值操作不改变右侧的任何值，只是与变量名建立关联关系。比如：

```
x = 1
x = x + 1
```

这个例子中的第二行代码在数学中可能意义不大，但是在程序语言中却很有用，比如实现累加的效果。

2.2.2　变量名和值的关系

在程序中我们通过变量名访问变量值，表面上看，Python 中的变量名好像是它所对应数据的标签一样，这个标签贴到哪个数据上，它代表的就是什么类型的数据，换一种解释就是变量的值是可变的。不过，关于变量值可变这件事，在不同语言中是有区别的，比如在 C 语言中，如果定义了一个变量并且声明了类型，就不能把这个变量的值更改为其他类型，而在 Python 中却没有任何限制。在本书中如果没有特殊说明，提到变量时均以 Python 为准。

Python 中变量名和值是标签和数据的关系，即通过标签可以获取它所代表的数据。假设有一台点唱机，里面存放着成千上万首歌曲的数据，每首歌的信息量都很大，当你想点歌时，并不需要把整首歌的歌词都输入进去，你只需要直接输入歌曲的名字就可以找到歌曲所对应的数据，那么，歌曲名字就相当于标签，它所关联的实际就是点唱机里面歌曲的数据。

现在，可以联想一下标签和数据其实存在着很多种关系。比如点一首歌 *Hotel California*，点唱机就会找到这个名字所对应的数据，这就是一对一。然后有人点了《加州旅馆》，点唱机播放的其实跟刚才是同一首歌，这就相当于一个数据有两个标签，即多对一。为了提高音质，点唱机里歌曲的源文件被换成了高清的，歌曲的名字这个标签从原来的文件指向到了新的文件，这就是标签的可移动性。我们可以在交互环境下体验这些效果。

```
>>>x = 1
>>>y = 1
>>>print(x)
1
>>>print(y)
1
>>>id(x)
1792698496
>>>id(y)
1792698496
>>>y = 2
>>>id(y)
1792698512
```

这个例子中的 id()返回的是对象在内存中的地址，值得一提的是，在 Python 中一切都是对象(如果你想深入理解对象，在面向对象编程中将有解释)。本例中的两个对象就是 x 和 y 两个整数。在这里，我们看到 x 和 y 的 id 是完全一样的，这是 Python 为了提高内存利用效率，对于一些简单的对象，如一些数值较小的 int(整数)对象，Python 采取重用对象内存的办法，如 x=1，y=1 时，由于 1 作为简单的 int 类型且数值小，Python 不会两次为其分配内存，而是只分配一次，然后将 x 与 y 同时指向已分配的对象，这就相当于一个数据有两个名字。最后，我们给 y 重新赋值时，再获取的内存地址就变了，这就是标签可移动(变量可变)。

不过，仅通过 id 返回的内存地址来判断，有时会有一些疑惑，例如：

```
>>>x = 12345
>>>y = 12345
>>>id(x)
61818192
>>>id(y)
61817488
>>>x is y
False
>>>print(x)
12345
>>>print(y)
12345
```

表面上看，x 和 y 的值都是整数 12345，但实际在内存分配上却不是一个数据，也就是在内存中存了两个 12345。通过身份运算符 is 可以判断出来，is 的作用是判断两个对象是不是一个，如果是，则返回 True；如果不是，则返回 False。不过，程序会判断 x 和 y 的值是一样的，通过 x 和 y 都可以获取相同的数据，所以，在实际编程过程中不用太在意 id。

如果要考虑内存空间，可以这样做：

```
>>>x = 12345
>>>y = x
>>>id(x)
```

```
61817744
>>>id(y)
61817744
>>>x is y
True
>>>print(x)
12345
>>>print(y)
12345
>>>x = 56789
>>>y
12345
```

在这里,y 和 x 的 id 相同,通过身份运算符 is 判断为同一个对象。我们通过 y = x 对 y 进行赋值,实际是将 x 所对应的值赋值给 y。但是,不要误会,当我们给 x 重新赋值后,y 是不会跟着改变的,因为这只相当于把 x 这个标签移动到了新的数据上。如果想要实现 x 变化 y 也跟着变,我们将在数据类型列表的部分进行讲解。

2.3 编写可以跟用户互动的程序: 输入、处理和输出

通过变量,可以方便地对数据进行存储或者关联,也可以在程序内被调用。但是很多时候,针对用户的程序经常会由用户输入数据。比如,《英雄无敌》开始时需要玩家输入名字,运行中需要玩家输入行进的方向,当程序获取到玩家输入的数据时,再在程序中进行相应处理,为了让玩家有直观的、更好的体验,程序还会向屏幕输出各种信息。这就涉及几乎所有有用的或好玩的程序都会有的三个特征: 输入、处理和输出。


编写交互式程序

下面通过编写一个简易计算器程序学习这三个基本要素。在这个例子中,我们只最小限度地解决问题,因为实际的计算器程序要更复杂一些。

(1) 输入: 用户输入的数字和运算符。

(2) 处理: 程序获取用户的输入并进行相应的运算。

(3) 输出: 程序向屏幕打印提示和结果。

注意: 在刚开始学习程序设计时,可以先最小化需求,也就是实现最基本的功能,然后再不断完善。不要追求一下做到完美,因为在实际工作中,需求是不断变化的,最重要的是核心功能,否则就会每天疲于应付各种需求的变化。

找出了核心功能,用能想到的最简单的方式实现。至于更复杂的功能,或者一个真正的计算器,随着学习的深入,自然而然就能完成。

(1) 获取用户输入的数字和运算符。解决这个问题需要使用 input() 函数,这个函数的作用就是获取键盘输入的数据和运算符。

```
>>>first = input()
123
```

```
>>>print(first)
123
>>>second = input('second int:')  #括号内的字符串会被打印到屏幕上
second int:456
>>>print(second)
456
>>>first + second
'123456'
>>>type(first)
<class 'str'>
>>>type(second)
<class 'str'>
```

从运行结果上看，现在的结果是字符串拼接，而不是两个数字相加。出现这个结果正是因为 input()这个函数，它的作用是获取键盘输入，无论输入什么内容，保存下来的都是字符串。通过类型判断函数 type()可以获取数据的真实类型。

这里我们要获得的是整数，而不是字符串。通过 int()函数做类型转换。

```
>>>first = int(input())     #int()作用是将其他类型转换为整数
123
>>>type(first)
<class 'int'>
```

(2) 程序内部处理数据并输出给用户。有了数字，内部运算就是把想法通过代码实现的过程。在学习怎样判断之前，我们让程序直接进行加、减、乘、除运算。为了让用户有更好的体验，可以润色一下，比如等待用户输入数据时不是黑屏，打印结果时有一些提示，具体程序如下：

```
#jisuanqi-0.1.py
first = int(input('first:'))
second = int(input('second:'))

print('" + ", result is :', first + second)
print('" - ", result is :', first - second)
print('"*", result is :', first * second)
print('"/", result is :', first / second)
```

运行结果如下：

```
first:1
second:2
" + ", result is : 3
" - ", result is : -1
"*", result is : 2
"/", result is : 0.5
```

2.4 快速理解对象和类型：数字和字符串

前面我们提到，Python 中一切都是对象，那么，对这些值进行操作即是对这些对象进行操作。每个对象都有与之对应的类型，这些对象其实就是某种类型的一个实例。例如，1、2、12345 是整数(Int)类型的对象；3.1415926 是浮点数(Float)类型的对象；'hello' 'world' 是字符串(String)类型的对象；还有结构型数据、列表、元组、字典等。在后续章节中，会系统学习数据类型以及自定义的类。

初学 Python，明确对象的类型很关键，因为类型决定了对于这个类型的对象，有哪些操作是合法的、允许的，以及操作的结果是什么。

程序语言中的数据与现实世界是相通的，Python 提供了丰富的数据类型，初期我们用得最多的就是数字和字符串，因为 Python 中的数字类型与我们熟悉的数学概念是相对应的，所以可以通过数字做一些适应性练习。


数据类型：数字

与数学对应的，Python 3 的数字分为整数、浮点数(小数)和复数(虚部用 j 标识)。Python 2 还有一个长整型，不用太在意长整型，Python 3 将其取消是有道理的，因为长整型其实就是一串比较长的整数。

```
>>>x = 123        #整型
>>>y = 1.23       #浮点型
>>>z = 1+ 23j     #复数型
>>>type(x)
<class 'int'>
>>>type(y)
<class 'float'>
>>>type(z)
<class 'complex'>
```

前面在讲变量时已经定义了一些数值，不过，我们做运算时所获取到的值，Python 如何知道是数字 1 和 12345，而不是字符串'1'和'12345'呢？比如 input()获取到的数字组成的字符串。

聪明的你可能已经发现了，我在写代码的时候就已经有区别了，就是引号。不错，Python 就是通过这样一些符号来做区分的，你可以试着做一个加法看看类型不同所得到的结果。

```
>>>x = 1
>>>x + x
2
>>>x = '1'
>>>x + x
'11'
```

看出区别了吗？不加引号的加法运算，将 x 作为数字处理，得到的就是正常的数学运算结果，而加了引号的字符串'1'，加在一起构成的还是一个字符串，结果是把两个字符拼接

在一起。

2.5 运算符和表达式

通过前面数字的例子，我们已经看到了简单的算术运算，Python 其实就是一个计算器，只要你能想到的，它都可以完成。这并不是说数学家发明了 Python，而是因为数学真的无处不在，特别是编程领域一直与数学密不可分。平时总会有人问诸如英语不好能不能学编程这样的问题，其实相比英语，数学更加重要。毕竟程序语言用到的关键字很有限，完全可以用拼音当变量名字，可是数学却不能替代，解决数学问题与编程有着相同的逻辑思维关系。


数学运算

其实数学运算就是将各种元素通过运算符组成表达式，最终得到一个结果。Python 中的运算符分为算数、赋值、比较、逻辑等。表达式就是将不同的数据(包括变量、函数)用运算符按一定的规则连接起来的式子。

2.5.1 算术运算符

数学运算首要的就是四大基本运算加、减、乘、除了，分别对应的符号如下。

```
>>>1 + 1        #加
2
>>>5 - 3        #减
2
>>>1 * 2        #乘
2
>>>5 / 2        #除
2.5
>>>5 // 2       #整除
2
```

这个例子是在 Python 3 中的结果，如果是在 Python 2 中，有个陷阱就是 5/2 的结果为 2，它会认为整数做除法，结果也要是整数。在 Python 2 中还专门有一个模块用来解决这种问题，Python 3 已经不需要了，在 Python 2 中可以这样解决这个问题：

```
>>>5.0 / 2
2.5
```

除了加、减、乘、除，Python 还支持指数运算和求余，如下所示。

```
>>>2 ** 3      #2 的 3 次方
8
>>>8 %3        #8 模 3 余 2
2
```

2.5.2 赋值运算符

有了运算结果，我们会保留下来：

```
>>>x = 1 + 2 * 5
>>>print(x)
11
```

这里就相当于变量赋值，等号右侧先进行计算，最终结果赋值给左侧。同时也可以看到上面的运算优先级跟数学运算优先级是一样的，也可以：

```
>>>x = (1 + 2) * 5
>>>print(x)
15
```

另外，赋值运算符在程序语言中有一种有意思的用法，如下所示。

```
>>>x = 1
>>>x = x + 1
>>>print(x)
2
```

上述代码的意思是变量 x 自身加 1，即自加，自加在 Python 中有专门的运算符+=，如下所示。

```
>>>x = 1
>>>x += 1
>>>print(x)
2
```

除了自加，还有自减、自乘、自除……如下所示。

```
>>>x = 10
>>>x -= 6
>>>print(x)
4
>>>x = 10
>>>x *= 2
>>>print(x)
20
>>>x = 10
>>>x /= 2
>>>print(x)
5.0
>>>x = 10
>>>x %= 3
>>>print(x)
1
```

```
>>>x = 2
>>>x * * = 3
>>>print(x)
8
```

2.5.3 比较运算符

在编程过程中，经常会通过比较一些数据来做判断，这时就会用到比较运算符，而且通常会配合判断语句使用，因为比较的结果只有成立和不成立，在编程语言中称为真和假，用 True 和 False 表示，True 和 False 属于布尔类型，术语叫布尔值。比较运算符用法如下。

```
>>>a = 10
>>>b = 20
>>>a == b          #a 是否等于 b
False
>>>a != b          #a 是否不等于 b
True
>>>a < b           #a 是否小于 b
True
>>>a > b           #a 是否大于 b
False
>>>a <= b          #a 是否小于等于 b
True
>>>a >= b          #a 是否大于等于 b
False
```

2.5.4 逻辑运算符和布尔值

在比较运算符中，我们知道了布尔值，其实，除了 True 和 False 还有其他类型的值，计算机也会处理为 True 和 False，比如 None、0、所有空的值(空字符串、空列表等)都为 False，其余均为 True。

逻辑运算符有三个，如表 2-2 所示(表中实例的前提为 a = 10，b = 20)。

表 2-2 逻辑运算符

运算符	逻辑表达式	描 述	实 例
and	x and y	布尔"与"：如果 x 为 False，x and y 返回 False；否则，返回 y 的计算值	(a and b) 返回 20
or	x or y	布尔"或"：如果 x 是非 0，返回 x 的值；否则，返回 y 的计算值	(a or b) 返回 10
not	not x	布尔"非"：如果 x 为 True，返回 False；如果 x 为 False，返回 True	not(a and b) 返回 False

2.5.5 成员运算符

Python 还支持成员运算符，用于判断序列中是否存在指定的值，用法如下。

```
>>>'x' in 'xyz'        #'xyz'字符串中是否含有字符'x'
True
>>>'x' not in 'xyz'    #'xyz'字符串中是否不含字符'x'
False
>>>
```

2.5.6 其他运算符

除以上提到的运算符之外，Python 还包括在 2.2.2 小节中提到的身份运算符 is 和 is not(见表 2-3)以及以二进制方式运算的位运算符(位运算符本书不涉及，这里就不多介绍了)。

表 2-3 身份运算符

运算符	描 述	实 例
is	判断两个标识符是不是引用自一个对象	x is y，类似 id(x) == id(y)。如果引用的是同一个对象，则返回 True；否则，返回 False
is not	判断两个标识符是不是引用自不同对象	x is not y，类似 id(a) != id(b)。如果引用的不是同一个对象，则返回 True；否则，返回 False

2.5.7 运算符优先级

表 2-4 列出了从最高到最低优先级的所有运算符。

表 2-4 运算符及其优先级

运算符	描 述	优先级顺序
**	指数	最高
~ + -	按位翻转、一元加号和减号（最后两个的方法名为 +@ 和 -@）	↓
* / % //	乘、除、取模和取整除	↓
+ -	加法、减法	↓
>> <<	右移、左移	↓
&	位 'AND'	↓
^ |	位运算符	↓
<= < > >=	比较运算符	↓
== != <>	等于、不等于运算符	↓
= %= /= //= -= += *= **=	赋值运算符	↓
is is not	身份运算符	↓
in not in	成员运算符	↓
not or and	逻辑运算符	最低

2.6 如何快速获取帮助

在前面的章节介绍了如何提问和获取帮助，但远水解不了近渴。比如，这么多运算符，怎么没有正弦、余弦呢？别急，我们已经知道 Python 具有丰富的模块，关于数学运算就有一个 math 模块，math 模块中提供了丰富的数学运算方法，正弦余弦都是小意思。如果想知道模块中提供了哪些方法，可以通过 dir()函数得到：

```
>>>dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
```

这个办法可以直观地看到 math 模块提供的方法有哪些，你看到'sin'和'cos'了吧。

当然，你可能还需要其他方法，或者希望更详细地了解这些方法怎样使用，这时就可以使用 Python 中最强大的私人助手 help()了，用法如下。

```
>>>help(math)
Help on built- in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
    """
    省略几百行
    """
    cos(...)
        cos(x)

        Return the cosine of x (measured in radians).

    sin(...)
        sin(x)
```

```
        Return the sine of x (measured in radians).
    """
    省略几百行
    """

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    (built- in)
```

在这个显示结果中可以看到完整的帮助手册，其中省略的上百行内容都是关于方法的使用说明，可以看到 sin()和 pi，最后的 DATA 是属性。用的时候要注意，因为这些都属于 math 模块，所以调用时采用点分法，就像前面用过的 turtle 模块绘图：

```
>>>import math
>>>print(math.pi)
3.141592653589793
>>>math.sin(1)
0.8414709848078965
```

看到这里，你有没有明确一件事？当我们用到不熟悉的模块时，不必去网上查找资料，直接用 help()，基本就可以解决问题了。

注意：*使用 help()是初学者必须适应和习惯的事情，通常可以获得第一手资料，而且是最快捷、最权威的资料。*

2.7 彩蛋：打印正弦波

在学习基础内容时就能看到相关应用的最简示例对学习编程语言是非常有帮助的，要不然，你学了数字和运算总会问：这能干什么？就做个加减乘除？

打印正弦波这个例子涉及科学计算和数据可视化技术，说得高科技一点，跟深度学习还能扯上关系，按照忽悠性培训机构的说法"这节课我们学习人工智能的相关案例"。

这里我们需要用到两个模块：numpy 和 matplotlib。如果你的环境中没有，使用 pip 安装就可以了。

1. Python 的科学计算包——numpy

numpy(numerical python extensions)是一个第三方的 Python 包，需要额外安装(见图 2-1)，用于科学计算。这个库的前身是 1995 年就开始开发的一个用于数组运算的库。经过长时间的发展，基本上成了 Python 科学计算的基础包，当然也包括所有提供 Python 接口的深度学习框架。

安装及导入过程如图 2-1 所示。

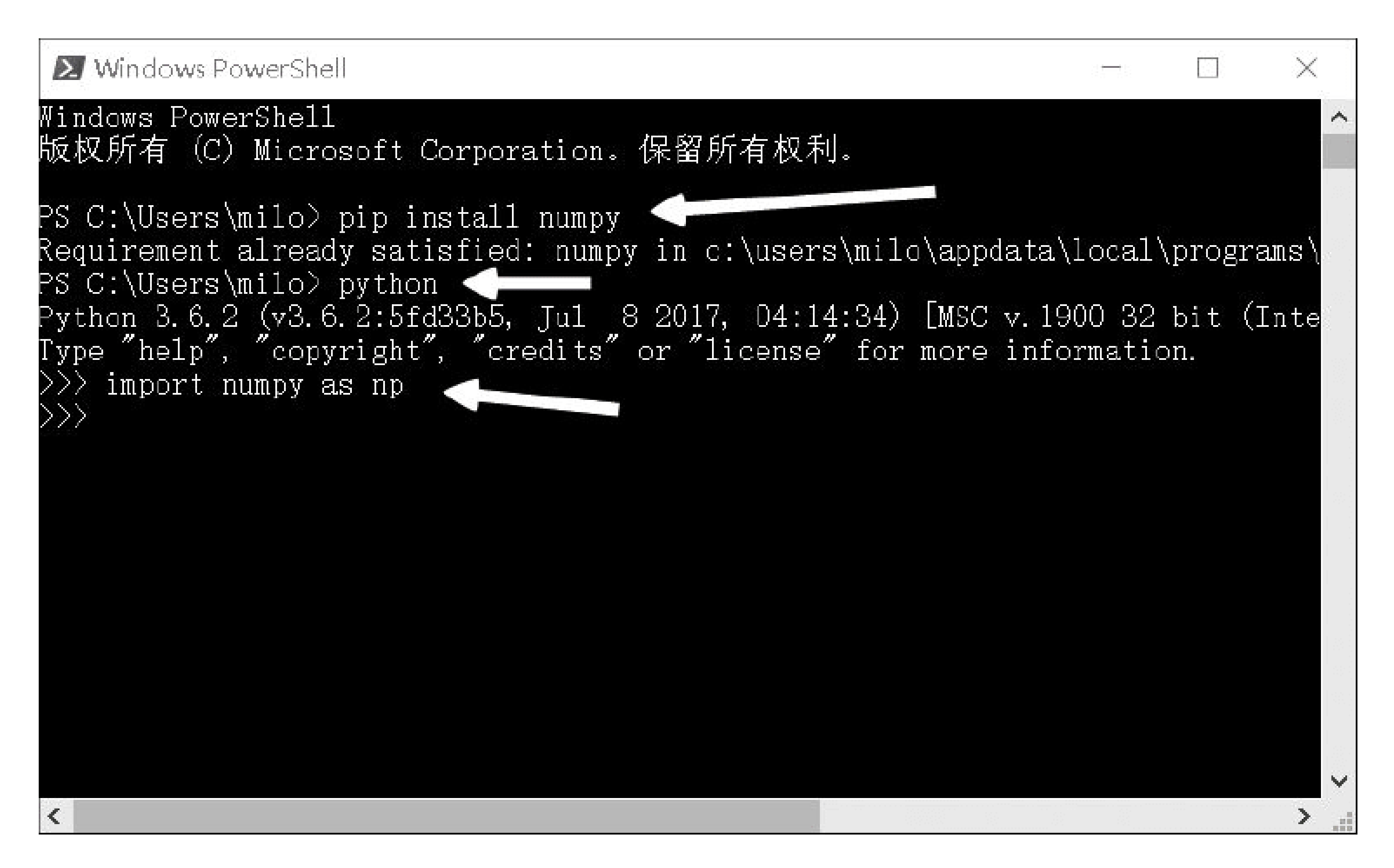


图 2-1 numpy 的安装及导入

注意:在导入 numpy 时,可以将 np 作为 numpy 的别名。这是一种习惯性的用法。

2. Python 的可视化包——matplotlib

matplotlib 是 Python 中最常用的可视化工具之一,可以非常方便地创建海量类型的 2D 图表和一些基本的 3D 图表。matplotlib 最早是为了可视化癫痫病人脑皮层电图相关的信号而研发的,因为在函数的设计上参考了 MATLAB,所以叫作 matplotlib。matplotlib 首次发表于 2007 年,在开源和社区的推动下,现在在基于 Python 的各个科学计算领域都得到了广泛的应用。matplotlib 的原作者 John D. Hunter 博士是一名神经生物学家,2012 年因癌症去世。

安装 matplotlib 的方式和 numpy 一样,通过 pip 安装就可以:

```
>>pip install matplotlib
```

3. 2D 图表

matplotlib 中最基础的模块是 pyplot。这里可用来画点图和线图,用法如下。

```
>>>import matplotlib.pyplot as plt
```

最终画正弦图的代码如下:

```
"""sin_0.1.py"""
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0,2 * np.pi,0.01)   #设定 x 的取值范围,从 0 到 2π,以 0.01 步进
y = np.sin(x)
plt.plot(x,y)                     #画模型的图,plot 函数默认画连线图
plt.show()                        #让画好的图显示在屏幕上
```

效果如图 2-2 所示。

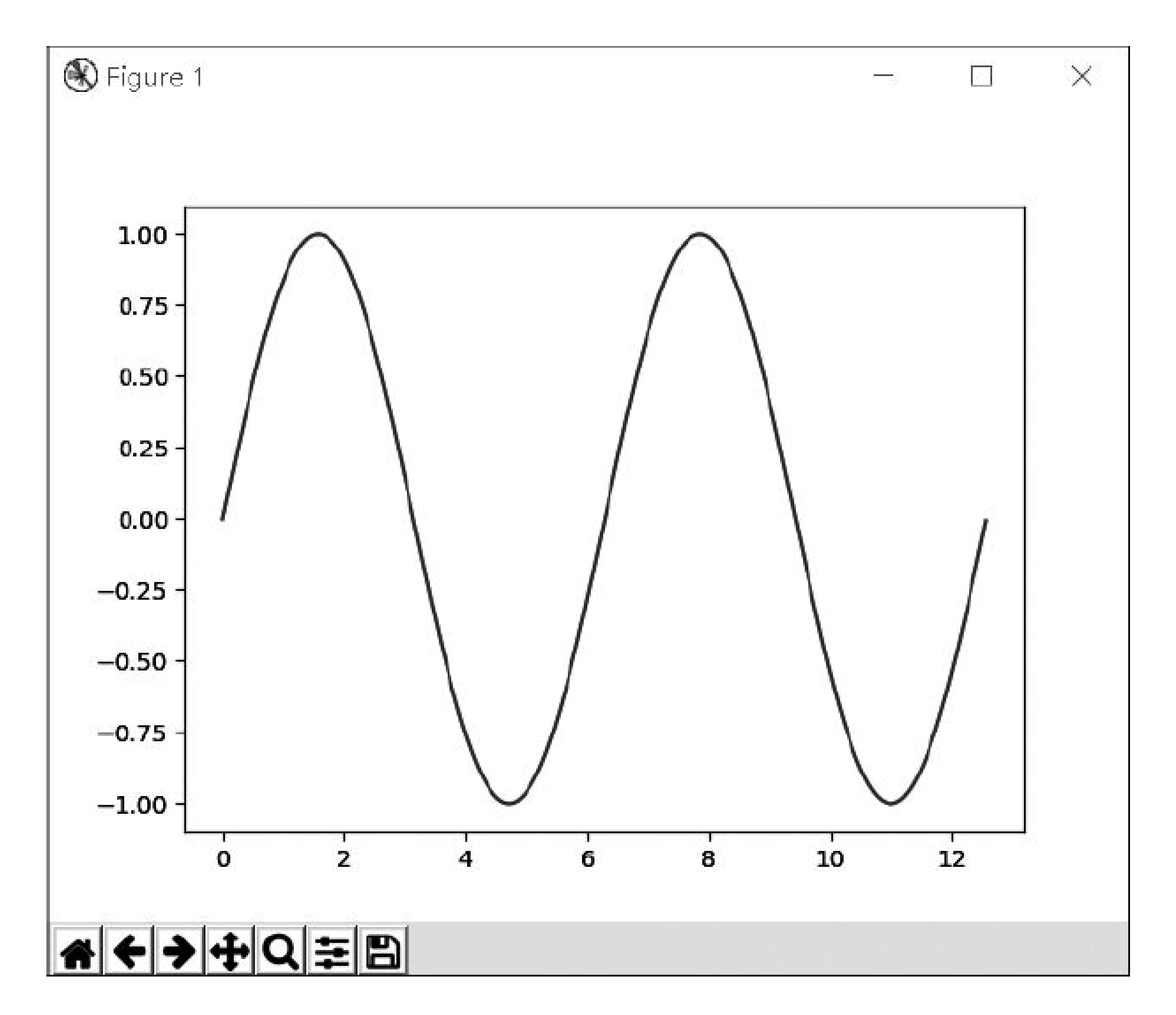


图 2-2　程序运行结果

这就是科学计算加数据可视化的一个简单实现，想做更多事，无非就是数据和方法的结合罢了。

重点提示

在这一章中，你要掌握的内容：

(1) 初步培养全局考虑问题的意识。

(2) 掌握变量的定义和数据类型的概念。

(3) 理解输入、处理和输出。

(4) 理解对象和类型。

(5) 了解运算符及使用方法。

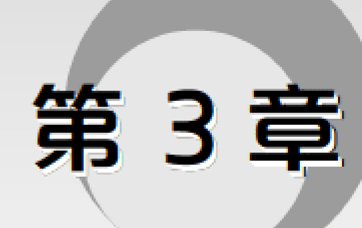
(6) 掌握快速获取帮助的方法。

动动手

(1) 以交互形式打印用户输入的个人信息，比如用户分别输入名字、性别、身高、年龄等，经程序处理后输出。

(2) 由用户输入两个数字，程序分别对这两个数字做加、减、乘、除四则运算，如果遇到屏幕上有错误提示，尝试找到原因(如果不太明白要做什么，可以学完下一章再做)。

(3) 书店所有书打 6 折销售，运费一本 6 元，每多加一本运费加一元，用户输入要买的书籍的单价以及购买的数量，屏幕输出总价。试编写程序实现。



第 3 章 搞定字符串

字符串是编程语言中用到最多的数据类型，许多实际问题的处理，最后都变成了字符串的问题。以《英雄无敌》为例，作为一个纯文字的游戏，若想让开场有些气势，运行的过程中更生动有趣，都需要围绕字符串来做文章。其他诸如网络数据传输、传递网址、发微博等，只要跟数据有关的基本都与字符串相关。

例如怎么实现图 3-1 所示的互动效果？ ♯号的方框是怎么出来的？

```
please input your name:milo
please input your password:123
############################
#welcome to milo's tool box#
# 1 : ls                   #
# 2 : free                 #
# 3 : quit                 #
############################
```

图 3-1　互动效果

代码如下。

```
print("############################")
print("#welcome to milo's tool box#")
print("#1 : ls                     #")
print("#2 : free                   #")
print("#3 : quit                   #")
print("############################")
```

上述代码可以实现图 3-1 的效果，但不止于此，关于字符串可以研究的太多了。接下来我们将介绍关于字符串的知识。

3.1 字符串的基本定义

字符串是一个序列，程序语言中认为可以打印的字符序列就是字符串。这个序列不一定是一个单词，它可以是一串密码 123456、一个网址，甚至是任意的组合，比如 abc、一篇完整的博客或者一个程序的所有源代码。



字符串定义和操作

在 Python 中构建一个字符串，或者定义一个字符串有两种方法。

一种是通过内建函数 str()生成，str()实际是类型内置函数，与整数 int()一样，用法如下。

```
>>>s = 123
>>>type(s)
<class 'int'>
>>>s + s
246
>>>s = str(s)
>>>type(s)
<class 'str'>
>>>s + s
'123123'
```

另一种更常用、也最直接的方法是通过引号定义，单、双引号都可以，只要成对出现就行，用法如下。

```
>>>h = "hello"
>>>w = 'world'
>>>hw = 'hello world"     #错误用法
```

Python 对字符串的单、双引号没有要求，但实际上双引号用起来会更方便一些，比如 "let's go"，如果使用单引号则需要对字符串中的单引号做转义处理'let\'s go'。

3.1.1 转义字符

在引号前加\，可将其后的符号转为普通字符处理，转义字符不会计入字符串，即打印'let\'s go'的效果是这样的：

```
>>>print('let\'s go')
let's go
```

利用转义字符可以实现更多的打印效果，表 3-1 为 Python 中的转义字符。

表 3-1 转义字符

转义字符	描 述	转义字符	描 述
\(在行尾时)	续行符	\n	换行
\\	反斜杠符号	\v	纵向制表符
\'	单引号	\t	横向制表符
\"	双引号	\r	回车
\a	响铃	\f	换页
\b	退格(Backspace)	\oyy	八进制数，yy 代表的字符，例如，\o12 代表换行
\e	转义	\xyy	十六进制数，yy 代表的字符，例如，\x0a代表换行
\000	空	\other	其他字符以普通格式输出

有了转义字符就可以打印这样的内容：

```
>>>say = "tom: \"let\'s go ! \" \njerry:'ok' "
>>>print(say)
tom: "let's go ! "
jerry:'ok'
```

3.1.2　Docstring

通过转义字符可以控制输出的样式，但是并不方便，试想要给一大段文章添加转义字符，那是很辛苦的。幸好在 Python 中还有一种选择，就是 Docstring，三种引号定义字符串。

通过一对三个单引号或者双引号定义字符串，效果即在编码时看到的就是输出效果，而不必再使用转义字符。前面例子的字符串可以这样定义：

```
>>>say = """tom: "let's go!"
jerry:"ok" """
>>>print(say)
tom: "let's go!"
jerry:"ok"
>>>say
'tom: "let\'s go!"\njerry:"ok" '
```

这里我们在定义字符串时就是按照显示的状态编写的，输出时保持了一样的效果，并没有使用转义字符，但实际在保存时，输入的回车、缩进等都会以转义字符的形式被保存下来。

三引号的好处除了可以让代码看起来更整洁以外，另外一个作用就是在前面讲注释时所提到的作用：可以通过三引号对程序做注释。

3.1.3　原始字符串

通过三引号字符配合可以定义出想要的字符串，不过有时我们希望转义字符被当作普通符号出现而不具备特殊含义，则需要将字符串定义为原始字符串，否则就很麻烦。我们在定义正则表达式时经常使用原始字符串，方法就是在字符串前加字母 r，用法如下。

```
>>>s = "a \n b"
>>>print(s)
a
 b
>>>s = "a \\n b"    #转义转义字符，多的时候操作不便
>>>print(s)
a \n b
>>>s = r"a \n b"    #定义原始字符串
>>>print(s)
a \n b
```

3.1.4 Unicode 字符串

字符串中如果含有中文,特别是不同的系统采用不同的中文编码,如果不是 Unicode 编码,也会比较麻烦,可能需要一次取两个或三个字符才能显示一个中文。所以如果涉及中文,建议将字符串定义为 Unicode 字符串,只需要在字符串前加 U 或 u,用法如下。

```
>>>hw = u"你好 hello"
>>>hw[0]
'你'
```

3.2 序列

字符串当中的字符是有序排列且顺序不可变的,其中每个字符对象都可以通过索引单独获取。同样具有这种特性的还有列表和元组,只是对象从字符串变成了更丰富的形式。我们先通过字符串讲解一下序列的特性和基本操作,列表和元组操作方法相同。



字符串——分片和索引

3.2.1 索引

索引(见图 3-2)可以理解为字符串中每个字符的编号,正序是从左到右,起始数字为 0;倒序是从右向左,起始数字为−1。

P	Y	T	H	O	N
0	1	2	3	4	5
−6	−5	−4	−3	−2	−1

图 3-2 索引

获取字符串内元素的方式:字符串对象后加方括号,方括号内加索引,用法如下。

```
>>>s = "PYTHON"
>>>s[0]
'P'
>>>s[3]
'H'
>>>s[-1]
'N'
>>>s[- 6]
'P'
>>>s[- 7]     #索引越界
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    s[- 7]
IndexError: string index out of range
>>>s[6]
```

```
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>      #出错代码位置
    s[6]
IndexError: string index out of range           #异常类型及提示信息
```

需要注意的是索引越界问题，超出索引界限就会抛出异常。这是本书讲解中第一次遇到异常，可能在前面的练习中你也遇到过，通常重要信息就是最后一行的异常类型和出错代码位置。通过异常信息，大多数时候我们可以很方便地找到问题所在。

注意：你可以在开始学习时试着记下这些异常类型，见得多了你就会发现，翻来覆去的就是那几个类型。异常是计算机跟你对话的途径，能够帮你快速发现问题，具体可以看第 10 章异常处理。

3.2.2　切片

切片也叫分片，功能相当强大，用于截取某个范围内的元素，通过[起始值：结束值]来指定起止区间(左闭右开区间，包含左侧索引值对应的元素，但不包含右侧索引值对应的元素)。这种切片操作很多时候会节省大量的代码，而且很方便，比如从一个长字符串中获取一个单词，用法如下。

```
>>>hw = "hello world"
>>>hw[0:5]
'hello'
>>>hw[:5]
'hello'
>>>hw[6:11]
'world'
>>>hw[6:]
'world'
```

上述例子中如果不指定起始值，则切片会取到起始位置；如果不指定结束值，则切片会取到结束位置。若都不指定，你自己试一下：

```
>>>hw[:]     #试一下，自己看结果吧
```

除了正向切片，负数的反向切片一样可以，上面的例子就变成：

```
>>>hw = "hello world"
>>>hw[-11:-6]
'hello'
>>>hw[:-6]
'hello'>>>hw[-5:-1]      #这个取不到结尾，想想为什么？ -1 换成 0 呢？
'worl'
```

在切片时加入第三个参数，作用是指定步长值，默认是 1，即前面的例子中没有指定步长值，步长值就都是 1。用法为

```
[起始值：结束值：步长值]：
```

```
>>>hw = "hello world"
>>>hw[::1]
'hello world'
>>>hw[::2]
'hlowrd'
>>>hw[::-1]
'dlrow olleh'
```

关于步长,你可以理解为在序列中取值时,有个小人从左向右走,它默认是走一步取一个元素,你指定了步长值后就会按指定的步长值运行,有意思的是步长值为－1 时,相当于改变了取值的方向,从右向左取。

在数字中使用步长,还可以实现获取奇偶数的效果:

```
>>>x = "0123456789"
>>>x[::2]
'02468'
>>>x[1::2]
'13579'
>>>x[::-1]
'9876543210'
```

有了切片功能,你可以用在《英雄无敌》里存储英雄的各项信息,用空格分开,然后通过切片获取其中的一部分,比如英雄的名字。当然,这样操作会很麻烦。解决的办法就是用列表存储数据,这个在第 5 章列表和元组再介绍。

3.3 与字符串相关的运算符

之前我们罗列了 Python 中常用的运算符,这一节讲其中涉及字符串的运算符。上一小节我们说字符串属于序列类型数据,其实针对序列类型数据的操作是通用的,也就是字符串和列表、元组的运算符是通用的,通过这些运算符,可以实现更丰富的序列操作效果。

3.3.1 拼接和重复

首先要讲的是两个数学运算符“＋”和“＊”。

“＋”作为连接符的作用我们在前面见过,它的作用就是将两个字符串拼接在一起。

“＊”是重复符,可以将原字符串重复指定整数次返回,需要注意的是,返回的是新字符串,原字符串不变,用法如下。

```
>>>name = "milo"
>>>hp = "100"
>>>hero = name + ' ' + hp
>>>hero
'milo 100'
>>>hp + 3          #不可类型混用
Traceback (most recent call last):
```

```
  File "<pyshell#16>", line 1, in <module>
    hp + 3
TypeError: must be str, not int
>>>3 * hp
'100100100'
>>>hp * 3
'100100100'
>>> (hp + ' ') * 3
'100 100 100 '
>>>hp
'100'
```

上面的例子中可以看到“＋”和“＊”的使用方法以及效果，需要注意的是“＋”两边的数据类型要一致，不要混合使用。

3.3.2 比较运算符

比较运算符也可以用于字符串中，当然，意义与数字中的比较运算符完全不同。同样，用于字符串的比较运算符也适用于其他序列类型数据。

比较两个字符串是否相同，使用“＝＝”运算符：

```
>>>'hello' == 'hello'
True
>>>'hello' == 'Hello'
False
```

比较两个字符串大小，用“＞”和“＜”运算符，但是，比较两个字符串的大小，情况比较复杂，主要分成两种情况：单字符比较和多字符比较。

1. 单字符比较

单字符比较的实质是两个字符 ASCII 码的比较。因为计算机存储数据时都是数字形式，而 ASCII 码的作用就是将字符映射到一个整数。可以通过 ord()和 chr()两个函数来了解 ASCII 编码与字符之间的关系。

```
>>>ord('a')
97
>>>ord('z')
122
>>>ord('A')
65
>>>ord('Z')
90
>>>ord('+')
43
>>>chr(99)
'c'
```

所以，比较两个单个字符时，就是比较通过 ASCII 编码的数字，你会看到 A 并没有比 a 大。

```
>>> 'a' < 'A'
False
>>> 'a' < 'b'
True
>>> '+' < 'A'
True
>>> '+' < 60
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    '+' < 60
TypeError: '<' not supported between instances of 'str' and 'int'
```

虽然 ASCII 码是把字符编码成数字，但不能直接用数字进行比较。

2. 多字符比较

多字符比较复杂些，但原理仍是基于 ASCII 码。比较的过程是并行检查两个字符串同一索引位置的字符。从索引为 0 的字符开始，然后由左向右，直到找到不同的字符为止。如果短字符串一直到结束为止都跟长字符串一样，则长字符串大。

```
>>> 'abc' < 'abd'
True
>>> 'abc' < 'zbc'
True
>>> 'abc' < 'abcd'
True
```

3.3.3 成员判断

in 和 not in：用于判断字符串的成员，判断运算符右侧字符串是否包含左侧字符串，用法如下。

```
>>> 'hero' in 'hero 100'
True
>>> 'x' not in 'hero 100'
True
>>> 100 in 'hero 100'      #抛异常的原因?
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    100 in 'hero 100'
TypeError: 'in <string>' requires string as left operand, not int
```

3.4 灵活多变的字符串操作

对字符串进行更多操作之前，我们要先搞清楚一些概念。

3.4.1 函数

函数是用来解决特定问题的一段小程序，函数程序都会封装起来，所以用户不用看函数的代码是怎么写的，只要知道函数怎么用就可以了。函数可以处理用户提供的数据，做处理后给用户提供返回值。函数的优点是减少重复的程序。

Python 提供了很多类型的函数，针对字符串的有很多，比如，可以通过 len()获取字符串的长度：

```
>>>len('hello')
5
```

Python 自带的函数叫作内建函数，len()函数不是某个类型专用的函数，相对来说是通用的，针对性不强，我们也可以用来获取其他类型序列的长度，比如列表的元组。其他相关的内建函数还有 max()、min()、sum()和 reversed()等，你可以试一下就知道用法了。

另外，我们也可以通过内建函数 str()将其他类型的数据转化为字符串，用法如下。

```
>>>numInt = 12345
>>>numFloat = 3.1415926
>>>numList = [1,2,3,4,5]
>>>str(numInt)
'12345'
>>>str(numFloat)
'3.1415926'
>>>str(numList)
'[1, 2, 3, 4, 5]'
```

上述转化很生硬，比如最后列表转化成了含有“[”“,”和“]”的字符串，如果要变成其他形式的字符串，就需要专门的方法了，将在 3.5 节和 3.6 节中进行讲解。

3.4.2 对象和方法


对象和方法

每个类都有对应的对象，比如数据类型，整数、浮点数、字符串都是值的类型，都是抽象的定义，简称为类。100、3.14、'hero'这些值我们可以称之为整数对象、浮点对象、字符串对象，这些对象是具备了其对应类的全部特征的一个实例。

类在定义时，会定义一些功能，方式跟函数一样，这些功能可以被类实例化出来的对象所调用，称为方法。所以，方法的实质就是函数的另一种形态，区别在于，方法只能被这个类的对象所调用，对象通过点标记调用方法，比如让字符串 hello world 首字母大写，代码如下。

```
>>>'hello world'.capitalize()
'Hello world'
```

这里用到的就是字符串的 capitalize()方法。每个对象的所有方法可以通过 dir()和 help()看到,如下所示。

```
>>>dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__','__gt__', '__hash__', '__init__','__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

方法很多,不用刻意去背,随时可以查找,比如在 IDLE 中输入对象和点后,只要按 Tab 键就可以看到提示效果了(见图 3-3),这时列出的是所有方法按字母排序,用上下箭头进行选择即可。也可以多输入几个字母,比如 cap,这样就可以过滤掉无关的方法,只显示 capitalize。

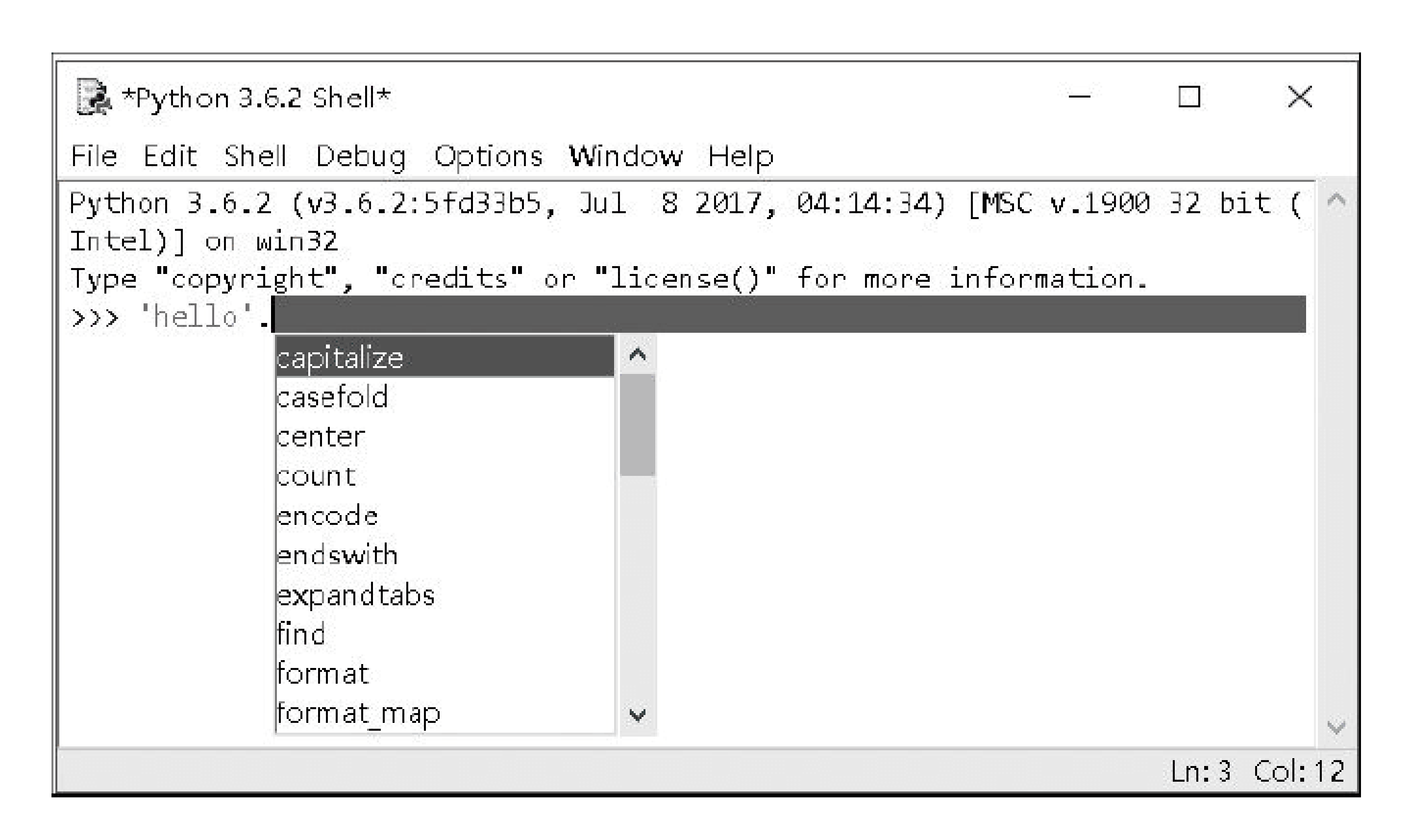


图 3-3 Tab 补全方法 1

如果想更快找到要用的方法,可以在点后输入前几个字母,比如要用 find()方法,就可以输入 f 再按 Tab 键,如图 3-4 所示。

其中的 find()方法,作用是查找指定字符串中是否含有给定的子串:

```
>>>hero = "player01 100"
>>>hero.find('player01')
0
>>>hero.find('100')
```

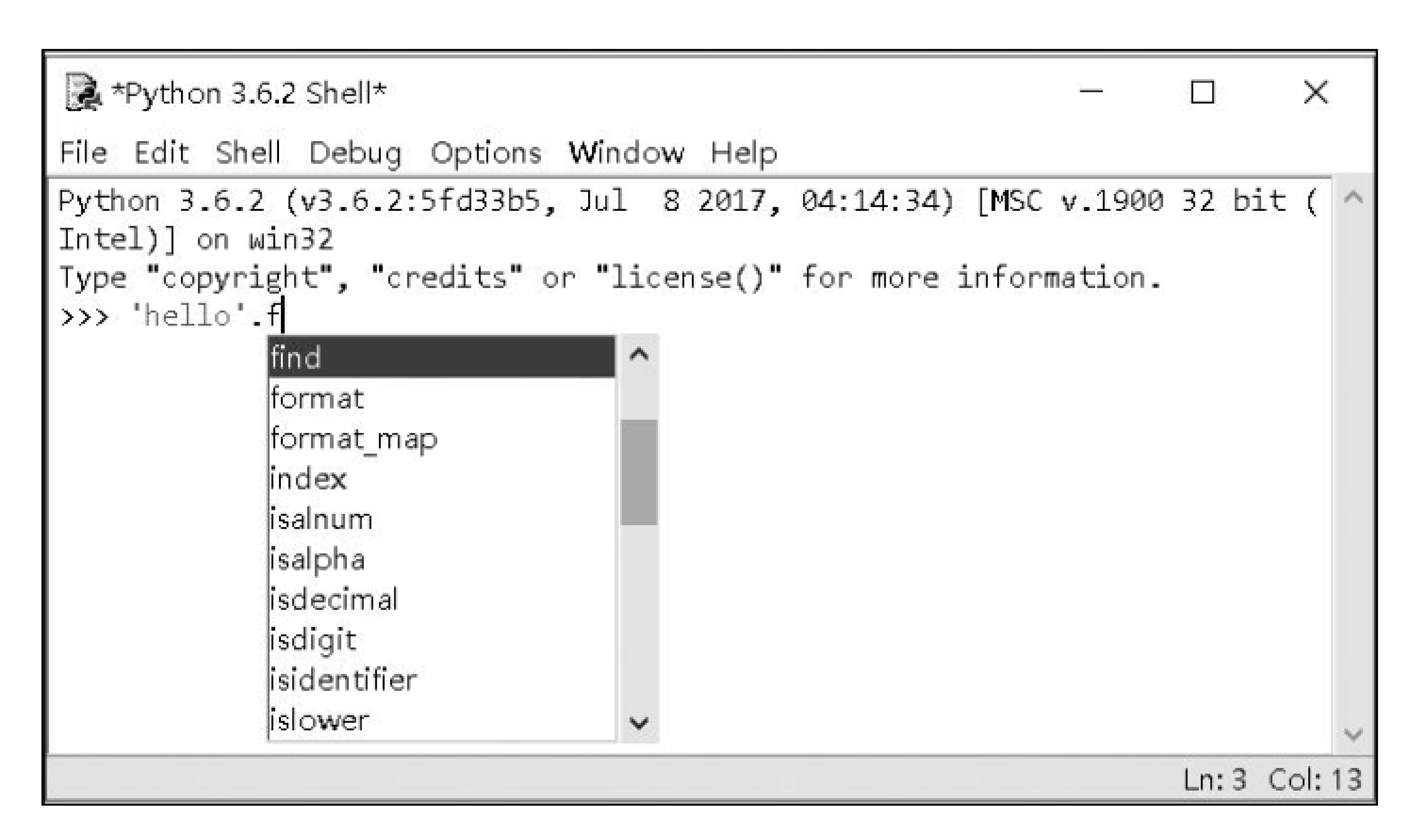


图 3-4　Tab 补全方法 2

```
9
>>>hero.find('0')
6
>>>hero.find('milo')
-1
```

如果找到了，则返回值是第一个字符的索引；如果没找到，则返回－1；如果有多个重复元素，则返回第一次出现的索引。如果想找到第二个或第三个，怎么办呢？

解决问题的思路：通常你在学习时并不能立刻掌握所有的知识点（事实上对于编程也没有必要），所以现在我们就一起来理顺一下解决这个问题的思路吧。首先，既然在字符串中查找子串返回的是索引，那么如果在查找时跳过不需要的索引开始查找是不是就行了呢？但是当我们并不知道有没有这样的方法的时候，可以从要操作的对象入手，查找是否有相关的方法。比如现在要查找字符串子串，可以先从 find 入手看看有没有其他用法，若没有其他用法，再考虑有无其他方法，从方法列表中还会发现一个 rindex()方法，然后通过 help()看一下具体的说明是否可以实现你的想法。下面就是这个过程。

首先从 find 入手：

```
>>>help(str.find)
Help on method_descriptor:

find(...)
    S.find(sub[, start[, end]]) -> int

Return the lowest index in S where substring sub is found,
such that sub is contained within S[start:end].  Optional
arguments start and end are interpreted as in slice notation.

Return -1 on failure.
```

这里的 str 代表字符串这个类型，并不是哪一个字符串。通过 help()可以发现，原来 find()还有可选参数，上述代码中的方括号即表示可选参数，这里的可选参数作用是指定查

找的起止索引范围，所以，只要跳过已找到的第一个元素的索引，再找到的就是第二个了。这个例子我们用了方法的嵌套，就是方法里面还有方法，执行顺序是先执行方法内的方法，并将返回值作为参数传递给上层方法。这并不高深，主要取决于方法所返回的值：

```
>>>hero = "player01 100"
>>>hero.find('0')
6
>>>hero.find('0',7)
10
>>>hero.find('0',11)
11
>>>hero.find('0',hero.find('0') + 1)
10
```

翻看其他方法会发现还有一个方法名字有 index 的 rindex()方法：

```
>>>help(str.rindex)
Help on method_descriptor:

rindex(...)
S.rindex(sub[, start[, end]]) ->int

Return the highest index in S where substring sub is found,
such that sub is contained within S[start:end].  Optional
arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.
```

通过 help()发现原来跟 find()查找方向相反是从右向左查找：

```
>>>hero = "player01 100"
>>>hero.rindex('0')
11
```

除了刚才用到的嵌套，还可以把方法和函数通过点标记连接起来使用，例如：

```
>>>hero = "Player01 100"
>>>hero.find('A')
-1
>>>hero.upper()
'PLAYER01 100'
>>>hero.upper().find('A')
2
```

3.4.3 分割和拼接

有时候需要从字符串或文件中提取一些有规律的数据，比如，有一个记录了很多人信息的文件，每一行存放了一个人的几项记录。以一行为例，这些记录都通过空格分隔，然后



我们要从中提取一些信息，比如这个人的名字，用法如下。

```
>>>milo = "milo 18 180 140"
>>>zou = "zouqixian 38 185 160"
>>>milo.split()
['milo', '18', '180', '140']
>>>zou.split()   #默认以空格为分隔符，返回值为一个列表
['zouqixian', '38', '185', '160']
>>>milo.split()[0]
'milo'
>>>zou.split()[0]
'zouqixian'
```

原本长短不一的字符串，经过分割变成了四个整体，这样就好操作了。再比如提取计算机的 IP 地址中最后一段的主机地址，用法如下。

```
>>>ip = "192.168.1.123"
>>>ip.split('.')  #自定义分隔符
['192', '168', '1', '123']
>>>ip.split('.')[-1]
'123'
```

现在我们可以把字符串分割成列表，反过来也可以把列表拼接成字符串，方法是以指定的字符串把列表的各个元素连接起来，用法如下。

```
>>>ip = ['192', '168', '1', '123']
>>>print(ip)
['192', '168', '1', '123']
>>>".".join(ip)           #以字符串"."调用 join 方法拼接列表中的对象
'192.168.1.123'
>>>print(".".join(ip))
192.168.1.123
>>>print("".join(ip))     #用空字符串拼接
1921681123
>>>print("aaa".join(ip))  #可自定义任意字符串
192aaa168aaa1aaa123
```

3.4.4 字符串模块

除了内置函数、方法之外，Python 还提供了一个 string 模块，用来提供更多的功能，我们先来看一下。

```
>>>import string
>>>help(string)
...
DATA
    __all__ = ['ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'cap...
    ascii_letters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
    ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
    ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    digits = '0123456789'
    hexdigits = '0123456789abcdefABCDEF'
    octdigits = '01234567'
    printable
    '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU...
    punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_'{|}~'
    whitespace = ' \t\n\r\x0b\x0c'
...
```

可以在其中看到一些属性是一些特殊的字符串，比如所有的小写英文字母、0～9 的整数、所有符号等。它们的用法也很多，比如要写一个程序，需要验证用户输入的是一个整数还是字母数字的组合、去掉字符串中的所有标点符号等。同时里面也提供了一些方法，用 help()就可以看到。

3.5 字符串格式化

我们通过 print()打印数据，但是 print()的功能比较单一，没有太多对于格式的控制，字符串的格式化则提供了更多控制格式的方式，使打印的效果更漂亮、工整。字符串格式化分格式化操作符(%)和内建函数 str. format()两种形式。

1. 格式化操作符（%）

%是 Python 风格的字符串格式化操作符，语法形式为

```
"Format %s string %d ..." %  (data1,data2,...)
```

格式化字符串在打印时与普通字符串一样，区别在于字符串中%s 和%d 这样的描述符位置会被%右侧括号内的一组数据所替换，而且数据和描述符的类型及数量都是对应的，例如：

```
>>>msg = "hello %s ,your hp is: %d"
>>>print(msg)
hello %s ,your hp is: %d
>>>print(msg % ('milo',100))
hello milo ,your hp is: 100
```

这个例子中的%s 是字符串描述符，表示需要用字符串来替换；%d 代表的是十进制整数，则需要用整数来替换。除了这两个之外，还有很多描述符以及可选项，其他可选项和常用的描述符如下：

```
%[(name)][flags][width].[precision]typecode  #方括号中为可选参数
```

其中，(name)——可选，用于选择指定的 key；

flags——可选，可供选择的值有：+表示右对齐，正数前加正号，负数前加负号；-表示



左对齐，正数前无符号，负数前加负号；空格表示右对齐，正数前加空格，负数前加负号；0 表示右对齐，正数前无符号，负数前加负号；用 0 填充空白处。

width——可选，占有宽度；

.precision——可选，小数点后保留的位数；

typecode——必选；

s 表示获取传入对象的__str__方法的返回值，并将其格式化到指定位置；

r 表示获取传入对象的__repr__方法的返回值，并将其格式化到指定位置；

d 表示将整数、浮点数转换成十进制表示，并将其格式化到指定位置；

f 表示将整数、浮点数转换成浮点数表示，并将其格式化到指定位置（默认保留小数点后 6 位）；

%，当字符串中存在格式化标志时，需要用 %%表示一个百分号。

通过宽度描述符 width 可以让数据变工整，就好像有个表格一样，用法如下。

```
#strformat.py
hero = "name:%-10s hp:%6d" #字符串宽度为 10 左对齐，数字宽度为 6
milo = ('milo', 100)
zou = ('zou', 6)
qi = ('qi', 99)

print(hero %milo)
print(hero %zou)
print(hero %qi)
```

运行效果如图 3-5 所示。

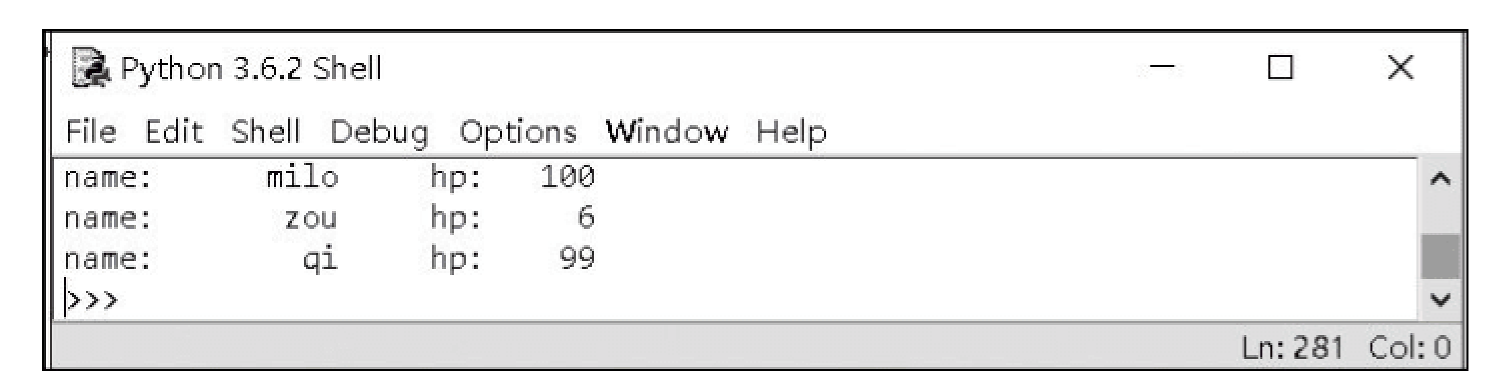


图 3-5　运行效果

.precision 是浮点数精度描述符，可以指定保留小数点后几位：

```
>>>import math
>>>math.pi
3.141592653589793
>>>print('pi = %.2f' %math.pi)
pi = 3.14
>>>print('pi = %10.2f' %math.pi)   #10 是宽度描述符(.2)是精度描述符
pi =       3.14
```

2. 内建函数 str.format

在 Python 中，字符串对象还有个方法 format，也是用来实现格式化字符串的。为了培养好习惯，建议一定要在手册中找到 str.format()，自学掌握，并且尝试自己学习消化。最

好不要用搜索引擎找别人实验的例子，那都是二手知识，相信自己，可能开始有点费劲，但学会查手册对于学习任何有文档的技术都是必备的。

3.6 遍历字符串

遍历可以理解为每个都访问一次的意思，如果想对字符串字符做逐个操作，就需要遍历。遍历操作需要 for 循环做迭代访问，最基本的方法为

```
>>>for i in "hello":      #i 是迭代变量，每次从字符串取一个字符
    print(i)

h
e
l
l
o
```

如果需要索引，可以借助一个函数 enumerate()：

```
>>>for i in enumerate("hello"):
    print(i)

(0, 'h')
(1, 'e')
(2, 'l')
(3, 'l')
(4, 'o')
```

enumerate()返回的是元组数据，每次返回一对索引和值，也可以直接把它们分开：

```
>>>for i,v in enumerate("hello"):
    print(i,"-->",v)

0 --> h
1 --> e
2 --> l
3 --> l
4 --> o
```

关于遍历的应用，在流程控制中还会在讲到。不过有了遍历，我们就可以对字符串做更多设计了，在作业中布置了一个小任务，去完成吧。

重点提示

在这一章中，你要掌握的内容：

(1) 熟悉各种字符串定义的方法。

(2) 理解序列的概念。



(3) 熟练字符串切片、拼接、成员判断、遍历等操作。

(4) 了解字符串操作的函数和方法。

(5) 熟悉字符串的格式化操作。



动动手

(1) 将姓名的中文表示换成英文表示，名在前，姓在后。

(2) 用户输入用“,”间隔的数字，如“1,3,5,23,45”，求出数字之和。

(3) 由用户随意输入一串字符，通过程序筛选出这一串字符串是否含有p、y、t、h、o、n六个字符。若有，拼接起来组成friend打印到屏幕上；统计出这一串字符可以组成几个friend。

(4) 设计一个简明的纯文字流程式游戏《英雄无敌》，随着学习的深入，会逐渐完善这个小游戏，你也可以根据自己的想法进行设计，大致流程如下。

① 游戏启动后显示欢迎界面。

② 玩家开始游戏前可以先给英雄起名字。

③ 游戏开始时先初始化英雄信息，如名字、血、攻击力等并显示到屏幕上。

④ 通过适合的方式存储人物名字、人物的血等。

⑤ 有个直线十格地图，英雄可在地图上一步一步前进(后期需要判断和循环)，暂时可以通过10个input()达到让用户输入10次数据的形式，从感觉上好像走了10步。

⑥ 在每一格上会有随机事件发生，如踩地雷掉血、吃包子加血等(后期需要函数封装)。

⑦ 随机事件用random模块或暂时可不用随机数，不同格发生固定事件即可。

第 4 章 流程控制

编程的目的是解决问题，现在我们已经可以让计算机按顺序执行(从上至下依次执行)指令了。但实际处理问题时，有些程序需要进行选择性执行，有些程序还需要反复执行多次。如果前面写的程序都是流水账，那么接下来我们掌握流程控制就可以随意控制程序的走向，写出千变万化的程序了。

没有流程控制的《英雄无敌》最多只是个按顺序执行的故事。有了流程控制，这个游戏就可以更加丰富灵活了，可以让玩家选择行进的方向，也可以让游戏一直运行到 Game Over 或者用户决定退出时才结束。

比如下面这样的运行过程("#"表示地图,"*"代表英雄所在位置)。

```
-*-welcome to Heroes world!-*-
input your name:milo
HI! milo You Hp is : 100

-*-the world is like this -*-
#######
-*-the '*' is you  -*-

contrl your hero:| 'a'for left | 'd' for right |
you are here *######
go or quit:d
you are h、ere #*#####
go or quit:d
you are here ##*####
go or quit:d
you are here ###*###
go or quit:d
you are here ####*##
go or quit:a
you are here ###*###
go or quit:a
you are here ##*####
go or quit:a
you are here #*#####
go or quit:quit
goodbey!!
```

4.1 让程序变智能的分支结构：if 语句

分支结构的语法并不复杂，经过精确的设计和组合，可以解决很多看似复杂的问题，比如人工智能领域的决策树算法就是由许多分支结构构成的，你可以让计算机用决策树算法分辨出一个物体是椅子还是狗。


布尔值和分支结构

if 语句是最基本的条件测试语句，通过条件表达式的结果选择执行指定的语句。判断对和错是计算机最擅长的事，因为理论上计算机只认识 0 和 1，而作为选择的判断依据就可以用 0(假)和 1(真)来表示。比如，英雄在地图上是否移动就是二选一，当然你可能需要上、下、左、右这种复杂移动的选择，即多个选择，这就需要将简单的选择进行组合或者使用字典(一种数据结构)来实现。

图 4-1 的意思就是，当主程序执行到 if 语句时就好像到了一个分岔路口，要根据一个条件表达式的结果作出判断。如果这个表达式的返回值是 True，则选择其中一条分支；如果是 False，则选择另一条，可以设置多个 elif 来增加条件，当通过判断后的语句块执行完毕后，则回到主程序的其他语句继续执行。

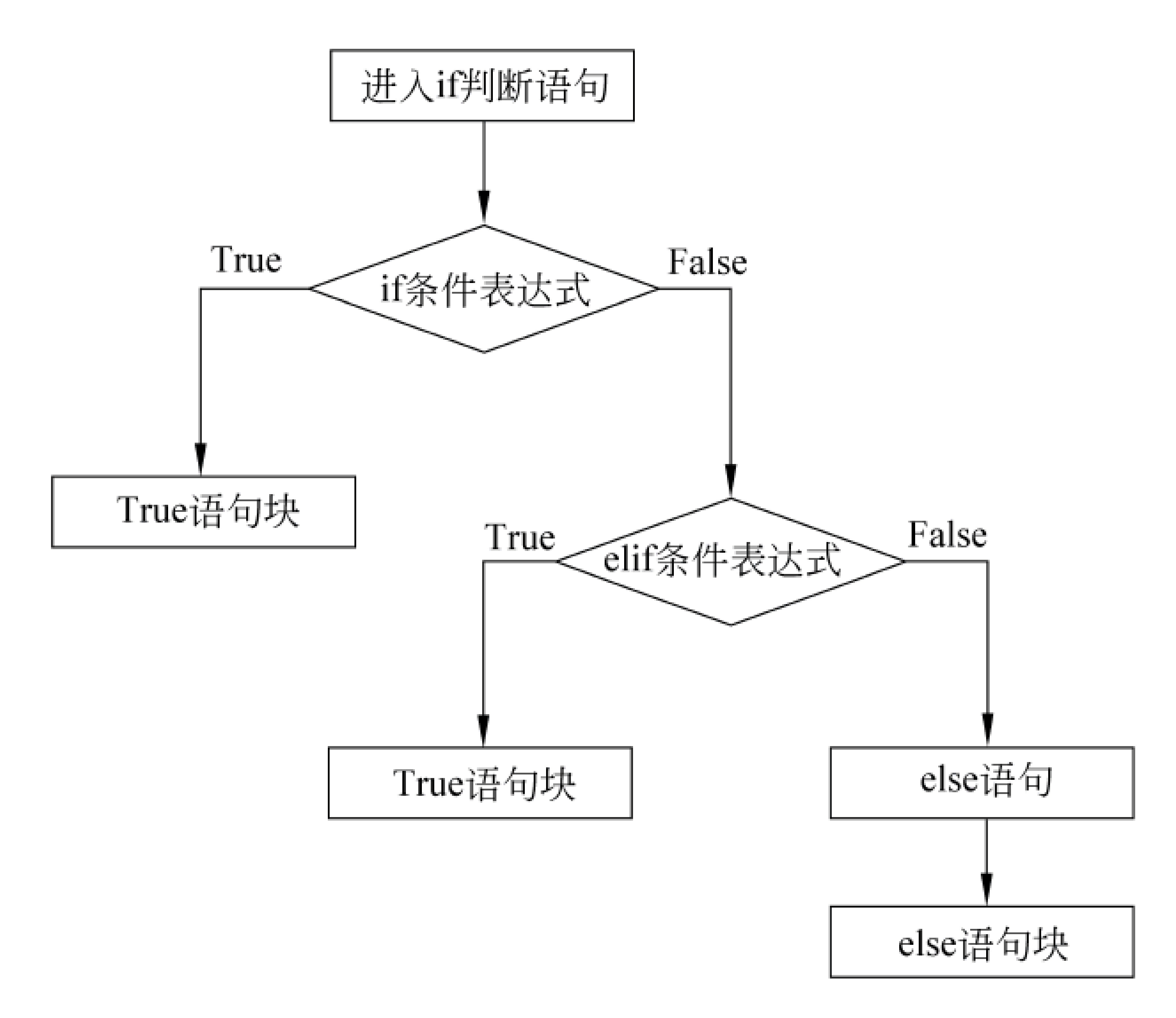


图 4-1 if 流程图

4.1.1 if 语法结构

if 的语法形式比较灵活，可以根据实际问题灵活运用，主要有以下四种形式。

1. 条件执行

最简单的条件执行，语法结构为

```
if 条件表达式:
```

```
    语句体
```

if 根据条件表达式返回的布尔值决定是否执行之后的语句体。布尔值为真，则执行语句体；为假，则什么都不做。为区别语句体与主程序，if 之后的语句体需要缩进 4 个空格，这种类型的语句成为复合语句。在 Python 中需要缩进的地方默认都是四个空格，当然，你也可以使用 Tab 键，但是最好设置 Tab 的缩进量为四个空格，而且千万不要空格和 Tab 混合使用，会造成很多麻烦。

语句体中的代码数量没有要求，有多行语句时需要保持一致的缩进量，但最少要有一行。如果在设计代码遇到一个语句体什么都不做（通常标记一个我们还没来得及写功能代码的时候），这时用 pass 语句，pass 语句的作用就是什么都不做，被称为代码桩，只起到占位的作用。

```
if age > 18:
    pass
```

2. 选择执行

第二种形式是选择执行，这种情况就会产生分支效果，仍然由条件表达式返回的布尔值决定被执行的语句体，语法结构为

```
if 条件表达式:
    语句体
else:
    语句体
```

这里多了一个 else，当 if 的条件表达式为 False 时，就会执行 else 的语句体。比如用程序处理一个申请，申请人需要输入自己的年龄，满 18 岁才可以通过，代码如下：

```
age = int(input("please input you age:"))
if age >= 18:
    print("Enter")    #此处行首有 4 个空格
else:
    print("Sorry! Too young!")
```

条件表达式 20>=18，返回值为 True，所以屏幕上会打印 Enter。

3. 条件链

有时需要判断的条件不止一个，需要更多分支。这时的语法形式叫条件链或者多分支：

```
if 条件表达式:
    语句体
elif 条件表达式:
    语句体
else:
    语句体
```

比如我们要做计算器，就需要判断用户输入的运算符，代码如下：

```
operator = input()
x = 1
y = 2
if operator == '+':
    print(1 + 2)
elif operator == '-':
    print(1 - 2)
elif operator == '*':
    print(1 * 2)
elif operator == '/':
    print(1 / 2)
else:
    print("opertor:+ - * /")
```

这里的 elif（其实就是 else if）意思是当上一个条件表达式返回值为 False 时，就会判断新的条件，使用的数量没有限制。所有的条件顺序执行，哪个条件为真，则执行哪个语句体，如果所有条件都为真，则只执行第一个。如果有 else 语句，只能放在最后。

4. 嵌套

条件判断语句体内可以再嵌套条件语句，以后我们再学其他语法，都可以灵活嵌套，目的是解决问题。

比如，做一个三分法，判断两个数字 x 和 y 的大小，代码如下：

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less then y')
    else:
        print('x is greater than y')
```

这个程序中的第一个 if 产生两个分支，其中第二个分支中又产生一个分支，区别的方式就是嵌套在内部的 if 相对于第一层整体进行了缩进。不过，嵌套语句层数不宜超过 3 层，嵌套层数太多会让代码阅读起来非常困难，应该尽量避免。

4.1.2 布尔值与 if

通过上面的例子可以看到分支流程二选一的过程，而其中的关键因素就是给定的条件表达式，想让程序“乖乖听话”，这个表达式要准确地反映出做判断的依据。

在条件表达式中需要理解的就是返回的布尔值，布尔值只有两个值：真和假。值得一提的是，虽然只有真和假，但表现形式不仅仅是 True 和 False 或者 1 和 0。

布尔表达式会被解释器看作 False 的值有：

(1) None。

(2) False。

(3) 任何为 0 的数字类型,如 0、0.0、0j。

(4) 任何空序列,如 ''、()、[]。

(5) 任何空字典,如 {}。

(6) 用户定义的类实例,如果类定义了__bool__()或者__len__()方法,并且该方法返回 0 或者布尔值 False。

其他所有值被解释器看作 True。

可以用 bool 将其他值转换为布尔值,在这里看一下效果,实际编程时不需要。

```
>>>bool('')
False
>>>bool('this is a test')
True
>>>bool(42)
True
>>>bool(0)
False
```

4.1.3 逻辑运算符与 if

有了布尔值,我们可以完成大多数的判断,但是碰到这样的问题呢? 比如报考 C1、C2 驾驶证的条件是满 18 岁小于 70 岁,这里有两个条件要同时满足才可以,这种情况我们有两种选择:一种是 if 语句嵌套实现,但是嵌套对于程序阅读来说不是很友好;另一种是利用逻辑运算符,第 2 章中我们简单介绍过逻辑运算符一共有三个 and、or 和 not。

1. 逻辑与

逻辑与运算符 and 的作用是,只有左右皆为真时结果才为真,例如:

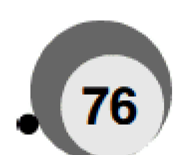
```
>>>True and True
True
>>>True and False
False
>>>False and False
False
```

前面提到报考驾驶证验证的例子可以写成这样:

```
age = int(input("please input you age:"))
if age >= 18 and age < 70:
    print("Enter")
else:
    print("Sorry!")
```

2. 逻辑或

逻辑或的作用是,运算符 or 左右有一个为真,则结果为真,跟逻辑与的区别在于:



```
>>>True or False
True
```

比如判断一个数是否为 3 或 5 的倍数,符合条件的如 3、5、6、9、15,条件语句如下:

```
x%3 == 0 or x%5 == 0
```

3. 逻辑非

逻辑非的作用就是给布尔值作否定,例如:

```
>>>not True
False
>>>not False
True
```

比如判断一个字符串不为空:

```
user_input = input("Your name:")
if not user_input:
    print("your name is: ",user_input)
```

需要说明一点,and 和 or 运算有一条重要法则:短路计算。

(1) 在计算 a and b 时,如果 a 是 False,则根据与运算法则,整个结果必定为 False,因此返回 a;如果 a 是 True,则整个计算结果必定取决于 b,因此返回 b。

(2) 在计算 a or b 时,如果 a 是 True,则根据或运算法则,整个计算结果必定为 True,因此返回 a;如果 a 是 False,则整个计算结果必定取决于 b,因此返回 b。

所以 Python 解释器在做布尔运算时,只要能确定计算结果,就不会继续往后计算,直接返回结果。

4.2 条件循环:while 语句

计算机可以在短时间内处理大量重复的语句,随着硬件技术水平的提升,计算机能完成的计算量越来越大,通过计算机来处理大量简单重复的指令可以完成很繁重的任务,这也是程序存在的意义。


循环

在编程领域里我们称重复为循环,从流程示意图 4-2 可以看出,执行重复语句的过程就像个循环的过程。Python 中提供了两种形式的循环语句来执行重复的指令,分别是 while 和 for。

4.2.1 while 语句

当要执行的重复指令与条件判断有关时,就要用到 while 了。while 语句是条件循环语句,如果条件为 True,就会一直重复执行循环体;如果条件为 False,则继续执行程序的其他部分。一般用来解决不确定循环多少次的循环,当然,通过程序设计也可以用 while 解决循

环次数确定的问题。

while 语句的基本语法为

```
while 条件表达式:
    循环体
```

while 语句的条件表达式的值是布尔型，循环体为 while 的子句，格式上与 if 类似，需要进行四个空格的缩进。

while 语句的执行过程如图 4-2 所示。

(1) 程序执行到 while 时，先判断条件表达式的值。

(2) 条件表达式值为真时，执行循环体代码。

(3) 循环体执行后 while 控制语句回到条件表达式位置继续判断。

(4) 如果表达式值为真，则重复(2)、(3)两个步骤，如果条件表达式值为假，则不执行循环体，同时 while 循环结束，程序继续执行 while 之后的代码。

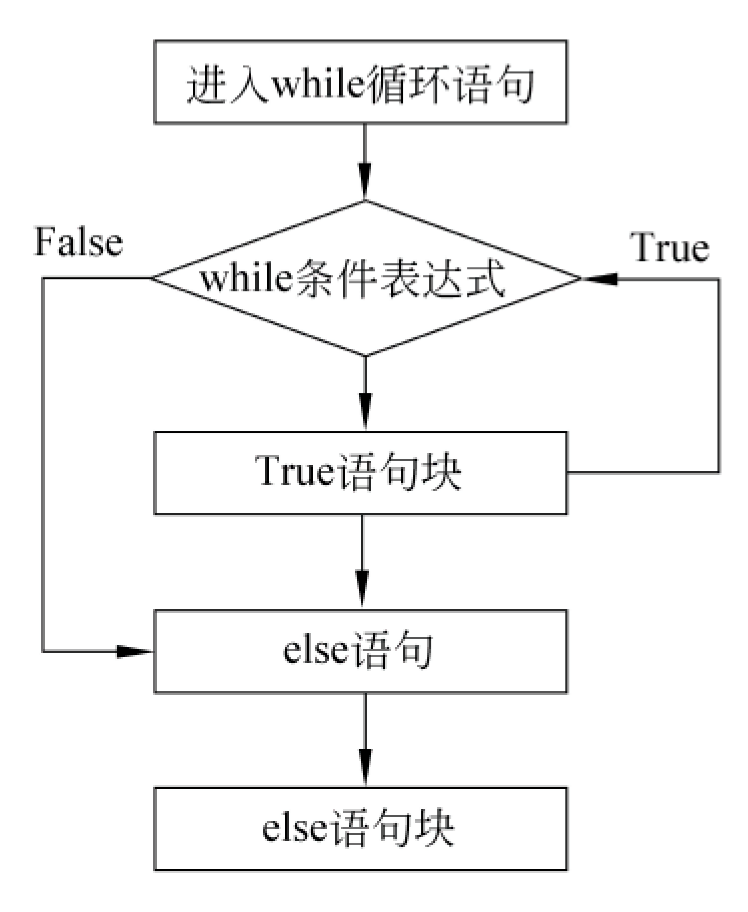


图 4-2 While 流程图

我们来看下面的例子，用户输入为 q 或 Q 时结束循环，否则一直执行循环：

```
#while01.py
user_input = input('input something:')

while user_input != 'q' and user_input != 'Q':
    print('your input is s%', user_input)
    user_input = input('"q" or "Q" for quit:')

print('here is not while')
```

程序第一行定义了一个获取用户输入的变量 user_input，while 语句根据这个变量值是否是 q 或者 Q 执行 while 的循环体，都不是，则条件表达式值为真，在循环体中，再次对 user_input 进行复制，这个值再带回条件表达式中进行判断，直到条件为假，while 循环结束。

运行效果如下：



```
>>>
input something:hello
your input is s% hello
"q" or "Q" for quit:quit
your input is s% quit
"q" or "Q" for quit:Q
here is not while
>>>
```

4.2.2 while...else 语句

另外，Python 中还提供了一种和其他大多数语言都不同的结构 while...else，基本语法为

```
while 条件表达式:
    循环体
else:
    语句
```

如果带有 else 语句，则 while 语句正常结束后会执行 else 字句；如果 while 语句被 break 破坏，则不执行 else 字句。break 是循环控制语句，用来终止循环。

我们改造一下上面的例子：

```
#while_else_01.py
user_input = input('input something:')

while user_input != 'q' and user_input != 'Q':
    if user_input == 'quit':
        break
    print('your input is s%', user_input)
    user_input = input('"q" or "Q" for quit:')
else:
    print('end of the loop!')

print('here is not while')
```

运行效果如下：

```
>>>
input something:hello
your input is s% hello
"q" or "Q" for quit:q
end of the loop!
here is not while
>>>
input something:hello
your input is s% hello
```

```
"q" or "Q" for quit:quit
here is not while
>>>
```

从结果可以看到，当通过条件表达式的值让循环结束时，else 语句内的代码会执行，而在循环内通过 break 指令结束循环时，则不执行 else 中的代码。

4.2.3 死循环和 break

设计循环的条件表达式时需要注意一点，就是可能会造成死循环。若条件表达式结果总是 True，就会一直进行循环体的执行。不过，有些时候我们会需要这种死循环，比如要一直执行一段程序，或者结束循环的条件比较复杂，条件表达式设计不便时。这时的用法通常如下：

```
while True:
    循环体
```

当然，自己设计的死循环一定要有中断指令，如 input()，或者可以结束的语句，如 break，否则，哪怕只是循环输出 hello world，对计算机来说也是无意义的资源浪费，不过，在交互模式下，可通过 Ctrl+C 组合键结束死循环。上面的例子就会变成这样：

```
#while_True_01.py
print('input something:')

while True:
    user_input = input('"q" or "Q" for quit:')
    if user_input != 'q' and user_input != 'Q':
        break
    print('your input is s%', user_input)

print('here is not while')
```

这种循环的形式很常见，因为这样我们就可以把循环的判断条件放在循环体内任何地方，并且可以表示当某个条件成立时结束循环，而不是在 while 顶端只能在条件失败时结束循环。

break 语句的作用是在循环过程中提前结束循环，通常通过 if 语句触发。有了 break，我们就可以灵活地结束循环，可以用在 while 和 for 循环中。但是，本着简洁的原则，如果只要很简单的条件就可以结束循环，就不要画蛇添足地设计 break 了。

4.2.4 确定次数的循环

有时候我们需要为循环执行指定次数，为了控制循环的次数，通常需要在程序中设置一个计数变量，每次循环时，这个变量自加或自减，当这个值达到指定值时，循环结束。

比如，想要计算 1+2+3+…+100 的值，就可以设计循环从 1 递增到 100 或从 100 递减到 1。因为这个值的范围是已知的，所以循环的次数也是确定的，程序如下：

```
#1+2+3+...+100
i, s = 1, 0
while i <= 100:
    s = s + i
    i += 1
print("1+2+3+4+...+100 = ",s)
```

运行结果如下：

```
1+2+3+4+...+100 = 5050
>>>
```

这个过程就是我们设计了一个计数变量 i 和一个累加变量 s，通过 i 的变化控制循环的次数，同时 i 作为每次累加的数值来源，最后通过 s 把每次的 i 累加起来。

需要说明的是，虽然可以这样设计一个固定次数的循环，但通常会用 for 来设计这样的程序。

掌握了 while 的语法，就可以结合 if 在开篇《英雄无敌》游戏中，实现英雄在地图上行走的效果了。具体的设计由你自己来考虑，不必完全一样。

4.3 迭代循环：for 语句

Python 中提供的另一个循环语句 for，与 while 根据条件表达式的真假确定是否进行循环不同，for 语句是通过迭代可迭代对象实现循环。for 接受序列或迭代器作为其参数，每次循环取出其中一个元素，循环的次数取决于序列或迭代器中元素的个数。

for 语句的基本语法为

```
for 变量 in 可迭代对象:
    循环体
else:
    语句
```

for 语句的执行过程如图 4-3 所示。

(1) 程序执行到 for。

(2) 变量从 in 后面的可迭代对象中取值，取到值则执行循环体。

(3) 循环体每次执行完后，再重复步骤(2)，由此产生循环。

(4) 变量从可迭代对象中获取不到值时，则不执行循环体，循环结束。

(5) else 的作用同 while，也是可选项。

Python 中的内置数据类型列表、元组、字符串、字典、集合等都是可迭代对象，可以通过 for 语句进行迭代，所以我们经常会使用 for 来遍历这些类型的数据，比如遍历字符串：

```
>>>for i in "hello world":
    print(i)

h
```

```
e
l
l
o

w
o
r
l
d
>>>
```

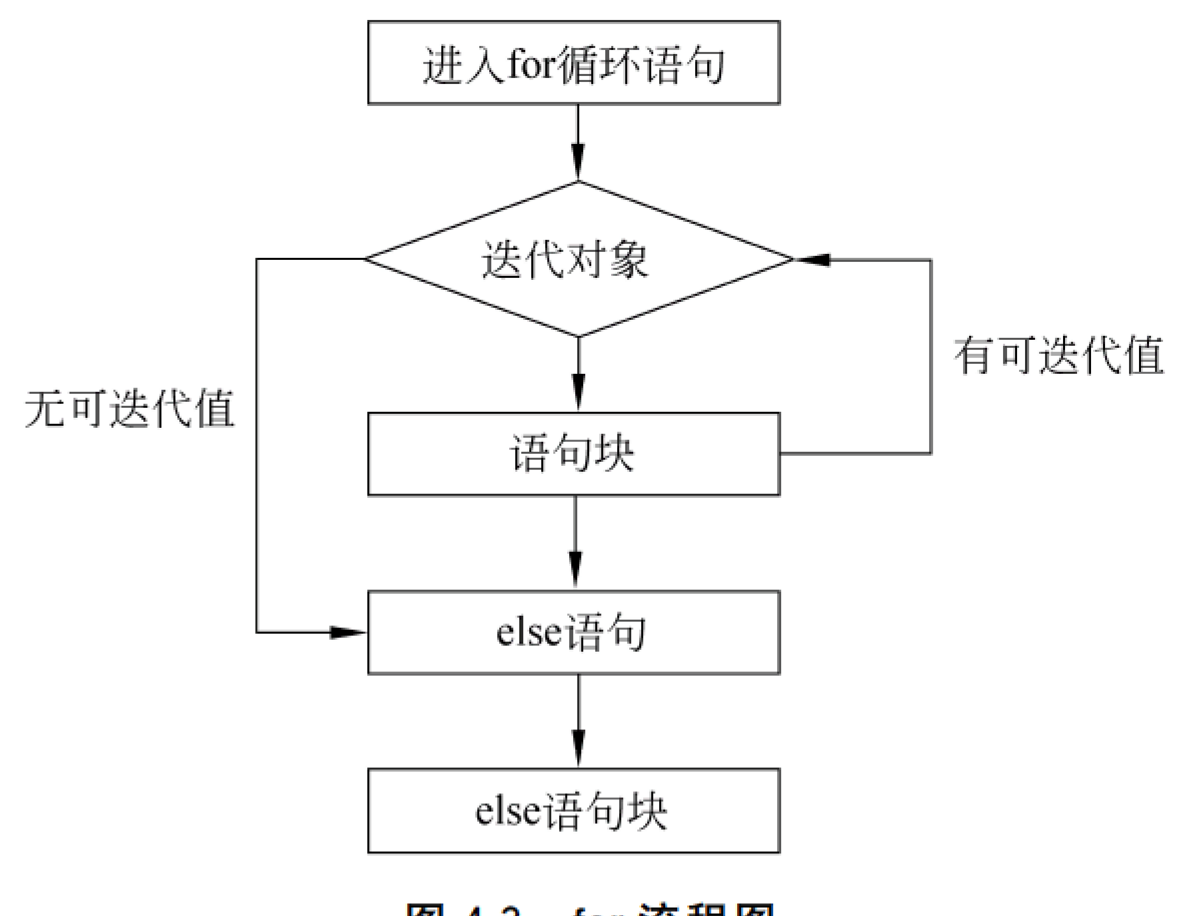


图 4-3 for 流程图

4.3.1 容器和迭代器

容器(container)是一种把多个元素组织在一起的数据结构,容器中的元素可以被逐个迭代获取,可以用 in 和 not in 关键字判断元素是否包含在容器中。通常这类数据结构把所有的元素存储在内存中(也有一些特例,并不是所有元素都放在内存,比如迭代器和生成器对象),在 Python 中,常见的容器对象有 list、set、dict、tuple、str。

我们也可以创建一个容器,包含一系列元素,可以通过 for 语句依次循环取出每一个元素,这种容器叫迭代器(iterator)。

迭代器简单来说就像机器一样生产数据,只是数据是事先订制好的,容器大多数时候是把数据放在内存中,而迭代器则是当有人去调用时才会在内存中记录一个值。实际的调用通过 next()方法。迭代器的意义在于当序列长度很大时,可以减少内存消耗,因为每次只需要记录一个值。有些时候只有迭代器是最好的选择。

一个简单的例子就可以看出区别,比如想循环一段程序一万次,如果用 for 循环来实现,你会用像刚才遍历 hello world 的例子那样用一个含有一万个字符的字符串来当作可迭代对象吗? 显然不会,这种情况通常用 range()函数,在 Python 2.7 中分 range()和 xrange()。它们的区别就在于,range()会直接在内存中生成一个列表,而 xrange()则是一个迭代器,当元素比较多时用 xrange()更省内存,所以在 Python 3 中 range()的功能直接变成 xrange(),

而不再分成两个。

range()的作用可以理解为可生成一个序列的迭代器,可指定起始值、终止值和步长值。

```
>>>i = range(10)
>>>type(i)
<class 'range'>
>>>print(i)
range(0, 10)
>>>for x in i:
    print(x)

0
1
2
3
4
5
6
7
8
9
>>>list(i)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

在这个例子里,我们通过 range()声明了一个可以产生 0~9 十个数字的迭代器,直接打印是看不见的,但是通过 for 就可以迭代出每一个值,当然,也可以直接把它转化成一个列表,在 Python 2 中比较 range()和 xrange()可以看得比较直观。

range()还可以通过参数生成一些特定的序列,例如:

```
>>>list(range(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>list(range(1,10,2))
[1, 3, 5, 7, 9]
```

至于迭代器本身,实质是实现了 next()方法的对象,常见的元组、列表、字典都是迭代器。迭代器中重点关注以下两个方法就可以了(iter 和 next 前后都是双下划线)。

__iter__方法:返回迭代器自身。可以通过 Python 内建函数 iter()调用。

__next__方法:当 next 方法被调用时,迭代器会返回它的下一个值,如果 next 方法被调用,但迭代器没有值可以返回,就会引发一个 StopIteration 异常。该方法可以通过 Python 内建函数 next()调用。

```
>>>i = iter("12345")
>>>type(i)
<class 'str_iterator'>
>>>print(i)
<str_iterator object at 0x05F886B0>
>>>next(i)
```

```
'1'
>>>next(i)
'2'
>>>for x in i:
    print(x)

3
4
5
>>>next(i)
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    next(i)
StopIteration
```

在这个例子中,我们把字符串“12345”变成了一个迭代器,这样,再直接打印就看不到值了,因为这些值没在内存上,我们可以通过 next()依次获取里面的值,也可以通过 for 去遍历,当遍历过后,实际已经去除了所有值,所以再次通过 next()取值的时候就会抛出异常 StopIteration 停止迭代。

4.3.2 实例:斐波那契数列

斐波那契数列(Fibonacci sequence)又称黄金分割数列,因数学家列昂纳多·斐波那契以兔子繁殖为例子而引入,故又称为“兔子数列”,指的是这样一个数列:1、1、2、3、5、8、13、21、34……这个数列从第 3 项开始,每一项都等于前两项之和。别小看这个看似简单的数列,在现代物理、准晶体结构、化学等领域,斐波那契数列都有直接应用。

用程序实现的方式很多,其实只要想清楚核心算法就简单得多了,纯算法的解决方式如下:

```
#feibo_while.py
a = 0
b = 1
i = 0
num = int(input())
while i < num:
    print(a)
    a, b = b, a + b     #核心算法
    i += 1
```

不过,相比较而言,使用迭代会更加高效,下面的例子会生成一个列表:

```
#feibo_for.py
numList = [0,1]
num = int(input())
for i in range(num -2):
    numList.append(numList[-2] + numList[-1])

print(numList)
```

4.3.3　循环嵌套


嵌套和循环控制

for 循环中也可以嵌套 if 语句，实现更灵活的程序。循环的嵌套通常是指在一个循环中完整地包含另外一个完整循环，也就是循环体中还有循环。while 和 for 可以相互嵌套。

比较典型的循环嵌套例子就是通过 for 循环的嵌套打印九九乘法表，写这个程序之前我们先分析一下：

(1) 九九乘法表由 9 行组成。

(2) 每行的列数有规律地递增一列。

(3) 表达式是 X * Y=Z。

(4) X 和 Y 的值分别由内循环和外循环控制。

为控制篇幅，下面为六六乘法表的代码：

```
#99.py
for i in range(1, 7):
    for j in range(1, i+1):
        print(j, '*', i, '=', i*j, '\t', end='')
    print('\n')
```

试着调整将其变成九九乘法表的代码吧。

这段程序中的逻辑关系：每次外循环 i 取值后进入内循环，内循环 j 的最大值等于 i，内循环全部结束后回到外循环 i 继续取下一个值，直到最后。

程序中为了控制打印效果，在内循环结束前不需要换行，而 print()默认是打印结束后换行的，也就是 end 的默认值是“\n”，设置为空则不换行，而每行结束后需要换行，所以有最后一行代码。可以多试试不同的打印效果，以便有更直观的印象。

运行结果如下：

```
1 * 1 = 1
1 * 2 = 2  2 * 2 = 4
1 * 3 = 3  2 * 3 = 6   3 * 3 = 9
1 * 4 = 4  2 * 4 = 8   3 * 4 = 12  4 * 4 = 16
1 * 5 = 5  2 * 5 = 10  3 * 5 = 15  4 * 5 = 20  5 * 5 = 25
1 * 6 = 6  2 * 6 = 12  3 * 6 = 18  4 * 6 = 24  5 * 6 = 30  6 * 6 = 36
```

4.3.4　循环控制 continue

除了前面介绍的 break 用来提前结束循环，有时只需要在某个条件下跳过当次循环，循环继续。Python 中提供的 continue 语句的作用就是：当循环遇到 continue 时，程序终止当前循环，并忽略 continue 之后的所有语句，然后回到循环顶端继续下一次循环。一定要注意 continue 和 break 的区别：break 是终止整个循环，continue 只是忽略当次循环的剩余语句。

打印所有 500 以内可以被 36 整除的整数，代码如下：

```
#continue36.py
s = 0
for i in range(1,501):
    if i%36 != 0:
        continue
    print(i,end = " ")
    s += 1
```

结果如下：

```
36 72 108 144 180 216 252 288 324 360 396 432 468
```

这段程序也可以改成不用 continue，请你试一下，会发现正好思路相反。



重点提示

在这一章中，你要掌握的内容：

(1) 会使用 if 语句配合各种布尔值和逻辑运算符完成复杂的分支语句。

(2) 明确 while 和 for 循环的用法及区别。

(3) 掌握循环的控制方法。

动动手

(1) 用 for 循环实现 1～100 所有数的总和。

(2) 已知 10 以内是 3、5 倍数的数字为 3、5、6、9，其和为 23。通过程序求出 1000 以内所有为 3、5 倍数的数字的和，参考结果为 233168。

(3) Collatz 猜想(冰雹序列)。

按下面的规则生成以 1 结束的序列：

若数字是偶数，除以 2；若数字是奇数，乘以 3 再加 1；当数等于 1 时，退出程序。

例如，从 5 开始，得到序列：5，16，8，4，2，1。

程序效果：用户输入一个数字，列出冰雹序列。

第5章 列表和元组

序列是程序设计中常用的数据存储方式，除了前面介绍的字符串(string)，Python 中提供的另外两种序列类型就是列表(list)和元组(tuple)，相比字符串，列表和元组能够存储的内容更丰富、更灵活多样。

Python 所提供的序列类型能够实现的功能在所有程序设计语言中是最强大的，有时甚至超出了简单的数据框架。

5.1 《英雄无敌》迭代开发：构建英雄世界



《英雄无敌》迭代开发：构建英雄世界

学习列表之前，先看一下在《英雄无敌》这个游戏中接下来要解决的问题。经过前面的学习，我们已经可以控制程序的流程了，但相信还有很多你觉得蹩脚或无法实现的效果。

现在先放下你要成为一个优秀程序员的想法，专注于一个程序的诞生过程。

首先，应该没有人像诗人一样突然灵感爆发一气呵成写出一个程序，即便有，在这之前也一定有过深思熟虑的积累才能爆发。通常程序开发的前期需要进行需求分析，比如《英雄无敌》这个游戏要实现的功能、要呈现的效果等，把它们列出来。

《英雄无敌》初步需求：

(1) 注册、登录、验证。

(2) 给角色起个名字，初始化英雄。

(3) 游戏的前奏。

(4) 满血出场。

(5) 有地图。

(6) 发生随机事件。

……

有了需求，就可以开始琢磨整个程序的框架了，如整个流程的控制、功能的设计等，最后才是开始编写代码，这时候代码可能已经在脑子里是个草稿了，写出来之后还要经过反复测试和调试。不过，即便可以成功运行，也不要高兴得太早，可能需求这时候又变了。程序写完了，需求又变了？嗯，程序员永远不知道客户和产品经理会有什么想法，这也是大家

经常开玩笑说程序员和产品经理“势不两立”的原因。还好现在不用担心，因为现在你是身兼所有职位在开发项目。那么在这个需求当中，列表和元组有什么作用呢？别急，往下看。

5.2 程序中的数据仓库：列表

作为序列类型的列表(list)，跟字符串相比，相同的是所有关于序列的操作都是通用的。不同的有两个方面：①字符串中的值只能是字符，在列表中值可以是任何类型，我们称列表中的值为元素或列表项；②列表是可变类型，即列表中的元素是可以改变的，甚至可以作为程序中的数据库使用，后面会有详细说明。


列表的创建和基本操作

明确了两者的异同，就可以套用已知的字符串知识快速掌握列表了。

5.2.1 创建列表

Python 中创建列表的方法很多，最基本的创建形式就是通过方括号[ ]，其中所有的元素通过逗号分隔开。另外，还可以通过 list()函数创建列表，具体看一下在 Python 会话中的几个示例：

```
>>>aList = []
>>>numList = [1,2,3,4,5]
>>>hero = ['milo',100,'hero']
>>>listInList = [1,2,3,['a','b','c']]
>>>hwList = list('hello world')
>>>hero
['milo', 100, 'hero']
>>>hwList
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>>type(aList)
<class 'list'>
>>>
```

上面的示例创建了几个形式各异的列表，分别来看一下：

(1) aList 是一个空列表，里面没有数据，不过这样的空列表在程序设计过程中会经常用到，比如有时候要通过循环或遍历构建一个新列表，这时就可以先建一个空列表，然后再把生成的值添加进来。

(2) numList 是一个纯数字的列表，所有整数用逗号隔开。

(3) hero 混合了两种类型的元素。

(4) listInList 是一个列表中又包含了另一个列表，这种我们称之为二元列表。

(5) hwList 是通过 list()函数将一个字符串转化成列表，每一个字符是一个元素，需要注意的是，非集合类型(数字、布尔值)的数据类型不能用 list()函数转化为列表。

5.2.2 列表拆分

列表可以通过赋值的方式进行拆分，用法如下：

```
>>>hero = ['milo',100,200]
>>>name, act, hp = hero
>>>name
'milo'
>>>act
100
>>>hp
200
```

5.3 列表的序列化操作

列表有非常丰富的相关操作，这也是列表功能强大的原因。不过，好在有些操作是序列通用的，比较好记，比如序列相关；有些功能性非常明显，比如数据的增删修改。下面先讲跟字符串相同的一些操作。

5.3.1 索引和切片

读取列表元素的方式就是变量名加索引（方括号中），因为列表与字符串同属序列，所以，在列表操作过程中也有索引和切片，而且用法完全一样，只是列表中的元素更丰富了。例如：

```
>>>hero = ['milo',100,'hero']
>>>hero[0]
'milo'
>>>hero[1]
100
```

通过索引，除了读取元素，也可以直接对指定索引的元素重新赋值。

二元列表的读取方式：

```
>>>listInList = [1,2,3,['a','b','c']]
>>>listInList[3]
['a', 'b', 'c']
>>>listInList[3][1]
'b'
```

这里，listInList 列表中有 4 个元素，其中第 4 个元素也是个列表。

需要注意的是，列表用方括号表示，读取索引也用方括号表示，例如：

```
>>> [1,2,3,4,5][-1]
5
```

若这个表达式弄明白了，说明你对列表已经理解了。这个例子将列表的创建和索引操作放在了一起，我们要从左向右阅读表达式，左边创建列表，右边是这个列表的索引。

最后，需要注意索引越界的问题，例如：

```
>>> [1,2,3,4,5][5]
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    [1,2,3,4,5][5]
IndexError: list index out of range
```

明确了索引的运用，接下来就是切片了，具体参照字符串中的切片操作，示例如下：

```
>>>hwList = list('hello world')
>>>numList = [1,2,3,4,5]
>>>hwList
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>>hwList[::2]
['h', 'l', 'o', 'w', 'r', 'd']
```

5.3.2 运算符及函数

字符串的重复(＊)、拼接(＋)、in 和 not in 在列表上的使用效果是一样的，示例如下：

```
>>>aList = [1,2,3]
>>>bList = ['a','b','c']
>>>aList + bList
[1, 2, 3, 'a', 'b', 'c']
>>>aList * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>3 in aList
True
>>>'d' not in bList
True
```

另外，也可以用＞、＜、＝＝、＜＝、＞＝和!＝ 比较两个列表，不过，因为列表元素类型比较多样，应使用同一类型的元素(全数字或全字符串)，避免不同类型的比较，示例如下：

```
>>> [1,2,3,4] < [1,2,3,4,5]
True
>>> [1,2,3,4] < [1,2,3,'a']
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    [1,2,3,4] < [1,2,3,'a']
TypeError: '<' not supported between instances of 'int' and 'str'
>>>
```

还有一些关于序列的通用函数是可以用于字符串、列表和元组的。

(1) len()函数用于获取序列的元素个数：

```
>>>len([1,2,3,4,5])
5
>>>len("abcde")
5
```

（2）max()函数和 min()函数返回序列中元素的最大值和最小值：

```
>>>max([1,4,6,8,5,3,2,9])
9
>>>min([1,4,6,8,5,3,2,9])
1
>>>max('12345')
'5'
>>>min('12345')
'1'
```

（3）sum()函数对数值型列表元素求和，非数值则报错：

```
>>>sum([1,2,3,4,5])
15
>>>sum([1,2,3,4,5,'6'])
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    sum([1,2,3,4,5,'6'])
TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

（4）sorted()函数对序列进行排序，reversed()函数可以直接将序列倒序排列：

```
>>>l = [1,4,7,6,2,9]
>>>sorted(l)
[1, 2, 4, 6, 7, 9]
>>>reversed(l)
<list_reverseiterator object at 0x060D7B10>
>>>next(reversed(l))
9
```

注意：reversed()函数返回的是一个迭代器，需要遍历或调用 next()函数。

5.3.3 遍历

在 for 循环中，已经介绍过通过 range()函数生成一个列表迭代器的方法，现在有了列表(列表跟字符串一样是序列)，当然就可以通过 for 直接遍历了(reversed()函数也可以)：

```
>>>for i in ['a', 'b', 'c']:
    print(i)

a
b
c
```

```
>>>for i in range(len(['a','b','c'])):
    print(i)

a
b
c
```

5.4 列表的操作

我们说列表时总会说它是可变的，跟字符串相比，其实就是指你不能把字符串指定索引位置的字符变成其他字符，而列表完全可以修改，甚至可以增加和删除元素，用起来就像数据库一样，而且也确实经常会把列表当作一个临时数据库来用。说临时是因为列表的增、删、改、查都只是在你的一个程序中，程序运行结束后并不会保存下来。



列表常用方法

5.4.1 可变的列表

列表的可变性非常有用，可以仅仅通过索引或切片来替换列表的元素甚至来个大换血，正因为这样，更要避免无意的操作带来的失误。下面是具体的改变列表的方式。

```
>>>role = ['milo',100,200,'hero']
>>>role[0]
'milo'
>>>role[0] = 'zouqixian'        #改变第一个元素
>>>role
['zouqixian', 100, 200, 'hero']
>>>role[-1] = 'monster'         #改变最后一个元素
>>>role
['zouqixian', 100, 200, 'monster']
>>>role[1:3] = [2000]           #第二到第三个元素替换为一个元素
>>>role
['zouqixian', 2000, 'monster']
>>>role[:] = [1,2,3,4,5,6]      #遍历列表替换为全新的元素
>>>role
[1, 2, 3, 4, 5, 6]
>>>role[:] = "hello world!"     #将列表指定元素替换为字符串字符
>>>role
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
>>>role[:] = 12345              #上个操作仅限可迭代数据，整数不可迭代
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    role[:] = 12345
TypeError: can only assign an iterable
```

在这个例子中，所有操作都是通过指定列表索引进行赋值实现的，每个操作都对列表的值进行了修改，即每次对列表标签（变量名）所对应数据空间的内部数据都进行了修改。

在这个过程中因为赋值语句不会有返回值，所以每次都通过变量名来查看结果。我们可以改变一个元素，也可以改变所有的元素，唯一需要注意的是被添加的元素必须是一个集合类型，集合中的每个元素作为元素单独添加到列表中。

5.4.2 列表的方法

针对列表的操作得益于列表对象的方法，就像字符串类型拥有大量的方法一样。列表的所有方法也可以通过 help(list)看到。我们按实现的效果来看一下。

1. 检索元素

通过索引获得列表的元素，通过列表的 index(x)方法可以返回 x 元素的索引，如果元素不存在，则报错。

通过 count(x)可以返回列表中 x 出现的次数，x 是一个元素。

```
>>>role = ['milo',100,200,'hero']
>>>role.index('hero')
3
>>>role.count(0)        #0 这个数字出现了 4 个，但是没有这个单独的元素
0
>>>role.count(100)      #100 出现了 1 次
1
>>>
```

2. 增加元素

append()方法可以向列表尾部添加一个新的元素：

```
>>>role.append('level2')
>>>role
['milo', 100, 200, 'hero', 'level2']
```

extend()方法可以将另一个列表的元素添加到当前列表(这个跟通过+号合并两个列表是有区别的，后面会详细讲解)：

```
>>>bag = ['AK47','knife',100]
>>>role.extend(bag)
>>>role
['milo', 100, 200, 'hero', 'level2', 'AK47', 'knife',100]
```

insert()方法可以根据索引将一个元素插入到列表的任何位置，这个方法需要两个参数，第一个是位置，即索引，第二个是需要插入的元素：

```
>>>role
['generral', 'milo', 100, 200, 'hero', 'level2', 'AK47', 'knife',100]
```

3. 删除元素

remove()方法可以删除一个指定值的元素，如果有多个，则从左至右依次执行：

```
>>>role.remove(100)
>>>role
['generral', 'milo', 200, 'hero', 'level2', 'AK47', 'knife', 100]
>>>role.remove(100)
>>>role
['generral', 'milo', 200, 'hero', 'level2', 'AK47', 'knife']
```

如果想弹出指定位置的元素，可以用 pop()，弹出的意思指删除的同时，这个值会返回给调用者：

```
>>>role
['generral', 'milo', 200, 'hero', 'level2', 'AK47', 'knife']
>>>role.pop(-2)
'AK47'
>>>bag = role.pop(-1)
>>>role
['generral', 'milo', 200, 'hero', 'level2']
>>>bag
'knife'
```

如果不需要返回值又想根据索引删除，可以使用 Python 的 del 语句，del 语句和切片结合就可以删除多个元素了：

```
>>>del role[0]
>>>role
['milo', 200, 'hero', 'level2']
>>>del role[1:]
>>>role
['milo']
```

5.4.3 字符串和列表

我们在 3.4 节介绍过字符串方法 split()，可以把字符串按指定参数(默认是空白)分隔并返回一个列表，以及通过字符串方法 join()将列表拼接成字符串。可以参考字符串章节相关内容，这里不再赘述。

5.5 Python 的魔术

先看一个过滤(filter)列表的例子。

已知 10 以内为 3、5 倍数的数字为 3、5、6、9，其和为 23，求 1000 以内所有为 3、5 倍数的数字的和。这个问题的解决方法很多，求 10 以内的，可采取下面的办法：

```
>>>numList = []
>>>for i in range(1,10):
    if i%3 == 0 or i%5 == 0:
        numList.append(i)

>>>numList
[3, 5, 6, 9]
>>>sum(numList)
23
```


列表解析和生成表达式

当然，这只是寻常的方法，下面我们要看看神奇的 Python 怎样求解 1000 以内所有为 3、5 倍数的数字的和。

5.5.1 列表推导式

列表推导式(list comprehensions)是 Python 中很强大的、很受欢迎的特性，具有语言简洁、速度快等优点。具体作用是通过一个序列生成一个新的列表。

列表推导式的语法为

```
[表达式 for 变量 in 列表]
[表达式 for 变量 in 列表 if 条件]
表达式:列表生成元素表达式,可以是有返回值的函数。
for 变量 in 列表:迭代列表将元素传入表达式中,如果有 if,则先交给 if 进行过滤。
if 条件:根据条件过滤
```

基本的用法举例如下：

```
>>>lst = [2,6,3,5,9]
>>>[x** 2 for x in lst]      #求所有元素的平方
[4, 36, 9, 25, 81]
>>>[x for x in lst if x%2 == 0]      #过滤出偶数
[2, 6]
```

由此，3、5 倍数的例子用列表推导式只要一行代码就可以了：

```
>>>[i for i in range(1,10) if i%3 == 0 or i%5 == 0]
[3, 5, 6, 9]
>>>sum([i for i in range(1,1000) if i%3 == 0 or i%5 == 0])
233168
```

怎么样，是不是看起来好像变魔术。

5.5.2 生成器表达式

生成器(generator)是一种特殊的迭代器，它的工作方式是每次处理一个对象，而不是一口气处理和构造整个数据结构，这样做的潜在优点是可以节省大量内存。

生成器表达式(generator expression)并不真正创建列表，而是返回一个生成器，生成器表达式语法结构和列表表达式一样，区别在于表达式使用()括起来，而不是[]。上面的例

子可以直接变成生成器表达式。

```
>>>lst = [2,6,3,5,9]
>>> (x * * 2 for x in lst)
<generator object <genexpr>at 0x061017E0>
```

这时可以看到返回值不是列表了,可以采取迭代的方式获取生成器的值:

```
>>>lst = [2,6,3,5,9]
>>>g = (x * * 2 for x in lst)
>>>type(g)
<class 'generator'>
>>>next(g)
4
```

3、5 倍数用生成器表达式就变成了

```
>>>sum((i for i in range(1,1000) if i%3 == 0 or i%5 == 0))
233168
```

可能会有人问只是换了个符号,意义是什么? 这就要回到迭代器上考虑了,迭代器最大的优点是可以节省大量的内存。

5.5.3 一点建议

虽然列表推导式和生成器表达式很方便,但并不是任何时候都可以用,有如下几个建议。

(1) 当只执行一个循环就可以时,尽量使用循环而不是列表解析,这样更符合 Python 提倡的直观性。因为列表推导式有的时候并不易读。

(2) 当有内建的操作或者类型能够以更直接的方式实现的,不要使用列表解析。例如复制一个列表时,使用 L1=list(L)即可,不必使用 L1=[x for x in L]。

(3) 如果需要对每个元素都调用并且返回结果时,应使用 L1=map(f,L), 而不是 L1=[f(x) for x in L],像 map()这样的函数还有很多,将在第 6 章介绍。

5.6 深拷贝、浅拷贝

我们一直在说列表是可变的,有时还希望对列表做备份,这就涉及拷贝,会有一些有意思或者让新手头疼的问题,在此,通过一些非常简单直观的例子来讲解。下面跟着我的思路做一遍应该就能明白了。

5.6.1 赋值

首先看一下字符串,来作为对比:

```
s1 = "hello world"
>>>s2 = s1
```

```
>>>s2
'hello world'
>>>s1 = ''
>>>s1
''
```

代码中定义了字符串 s1,然后 s2=s1,相当于 s1 的数据增加了一个名字,所以通过 s2 访问到的数据就是 s1 对应的数据,最后给 s1 赋值为空字符串,此时 s2 的值为

```
>>>s2
'hello world'
```

这里新手会犯一个错误,可能会认为 s2 会随着 s1 的改变而改变,但实际上 s2 的值并没有改变,原因在介绍变量时已经说过,s1 和 s2 都是数据的标签,当执行 s1='' 时,实际相当于将 s1 这个标签移动到了数据''上,而 s2 并没有移动,数据当然不会改变。

你可能会问,这有什么意义?接下来就是关于列表的可变特性,请看下面的例子:

```
>>>l1 = ['hello', 'world']
>>>l2 = l1
>>>l2
['hello', 'world']
>>>del l1[:]
>>>l1
[]
```

跟上面的例子类似,只是字符串变成了列表,并且删除 l1 所有元素,l2 的值为

```
>>>l2
[]
```

想到了吗?l2 的值也清空了,别急,再看一步:

```
>>>l1 = ['hello', 'world']
>>>l1
['hello', 'world']
```

l1 再赋值,l2 会怎样,同样获得值吗?

```
>>>l2
[]
```

依然是空。

什么原因呢?首先明确一点,在 Python 中,s1 = s2 和 l1 = l2 是一样的,都是给对象添加新的名字。但是为什么列表的值被改变的时候,另一个名字所对应的值也变了呢?这是因为创建列表时数据的存储方式,我们看一下 l1 = l2 之后的情况,如图 5-1 所示。

从这个结构表中可以看出,l1 和 l2 使用的是同一个存储空间,所以这时我们通过任何一个名字对值进行改变,另一方所访问的空间都不变,获取的值当然不变。而当我们第二

次执行 l1 = ['hello', 'world']时，实际上是将 l1 这个名字移动到一个新的列表对象上，如图 5-2 所示。

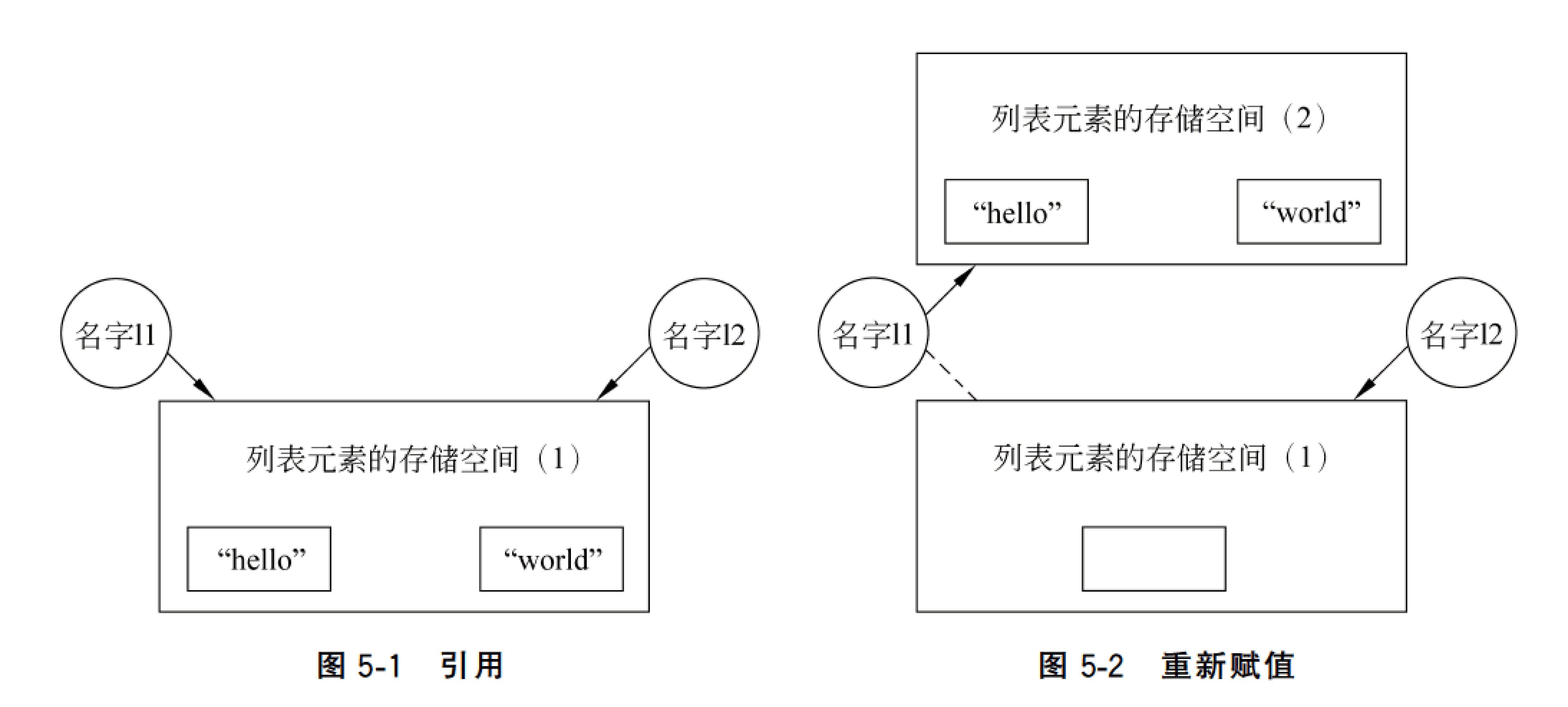


图 5-1　引用　　　　图 5-2　重新赋值

综上所述，关于 Python 中赋值操作的效果如下。

（1）赋值是将一个对象的地址赋值给一个变量，让变量指向该地址（旧瓶装旧酒）。

（2）修改不可变对象（str、tuple）需要开辟新的空间。

（3）修改可变对象（list 等）不需要开辟新的空间。

5.6.2　浅拷贝

Python 提供了一个 copy 模块，其中的 copy()方法可以实现浅拷贝，浅拷贝仅仅复制容器中元素的地址，效果如下：

```
>>>import copy
>>>a = ['hello',[1,2,3]]
>>>b = copy.copy(a)
>>> [id(x) for x in a]
[65592207, 5623045]
>>> [id(x) for x in b]
[65592207, 5623045]
>>>a[0] = 'world'
>>>a[1].append(4)
>>>print(a)
['world', [1, 2, 3, 4]]
>>>print(b)
['hello', [1, 2, 3, 4]]
```

在本例中，经过浅拷贝之后的副本，在做修改之前元素的地址是相同的，此时浅拷贝只是将源列表中元素的地址复制了一份，修改 a 列表内的不可变元素字符串时移动了地址，b 不跟着改变，而可变的列表元素则在内部增加了值，a 和 b 的这个列表元素地址相同，所以同时变化。

综上所述，浅拷贝是在另一块地址中创建一个新的变量或容器，但是容器内元素的地

址均是源对象元素的地址的拷贝。即新的容器中指向了旧的元素。

最后，除了 copy 模块，还可以通过切片操作、工厂函数 list()、对象的 copy()方法等实现浅拷贝：

```
>>>a = [1,2,3,4]
>>>b = a[:]
>>>c = list(a)
>>>d = a.copy()
```

5.6.3　深拷贝

深拷贝的方法是 copy.deepcopy()，完全拷贝一个副本，容器内部元素地址都不一样：

```
>>>from copy import deepcopy
>>>a = ['hello',[1,2,3]]
>>>b = deepcopy(a)
>>> [id(x) for x in a] [46952708, 66053004]
>>> [id(x) for x in b] [48657389, 66028736]
>>>a[0] = 'world'
>>>a[1].append(4)
>>>print(a)
['world', [1, 2, 3, 4]]
>>>print(b)
['hello', [1, 2, 3]]
```

在这个例子中可以很明显地看到，深拷贝后的对象地址和元素的地址都不同，可以看作是值相同的完全拷贝的副本，不论原对象怎么修改都不会影响深拷贝。

用图 5-3 示意浅拷贝和深拷贝。

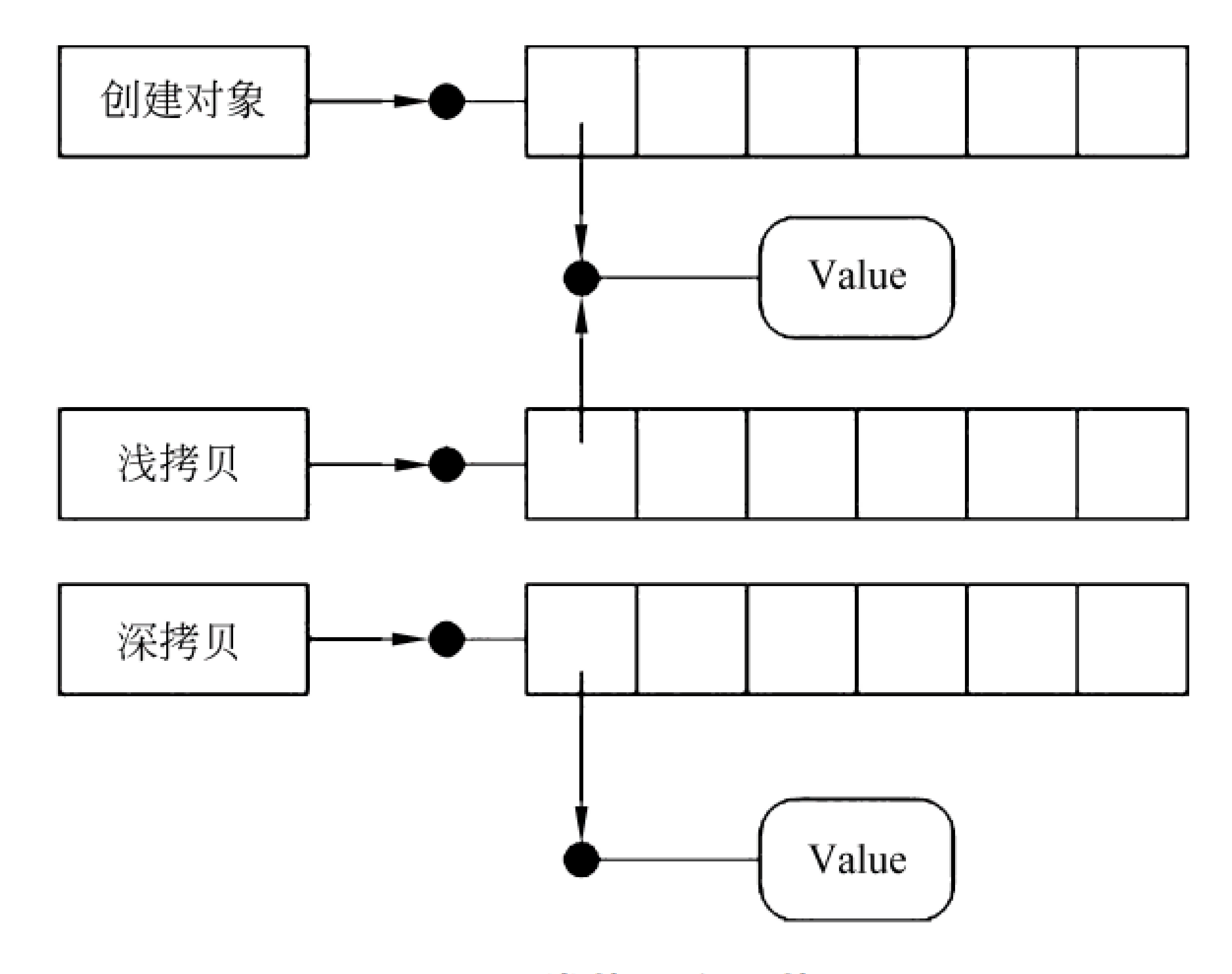


图 5-3　浅拷贝和深拷贝

5.7 不可变的列表——元组

元组可以看作是不可变的列表。元组几乎具备列表所有的特征，不同的是元组一旦创建就不可改变，不能更改元素，也不能增减元素。

因为元组跟列表相似，所以，除了更改元组元素，你可以尝试所有的列表操作，下面我们从区别于列表的一些特性了解一下元组。


不可变的元组和灵活的列表

5.7.1 创建元组

元组是用逗号分隔的一组值，这个值可以是任何类型的对象，创建元组时可以不用括号，但通常我们会用()将所有元素括起来，以区别于列表的[]，如下：

```
>>>t1 = (1, 2, 3, 'a', 'b', 'c')
>>>t2 = 1, 2, 3, 'a', 'b', 'c'
>>>t1
(1, 2, 3, 'a', 'b', 'c')
>>>t2
(1, 2, 3, 'a', 'b', 'c')
```

这里需要注意一点，介绍元组时经常会有一个误区，说元组的创建符号是()。实际上真正创建元组的运算符是逗号，Python 中圆括号大多数情况下表示分组，圆括号加上逗号才成为元组创建的一部分，比如创建一个只有一个元素的元组，若只有括号没有逗号，则跟元组一点关系也没有，如下：

```
>>>t3 = 1,
>>>t3
(1,)
>>>t4 = 1
>>>t4
1
>>>t5 = (1,)
>>>t5
(1,)
>>>t6 = (1)
>>>t6
1
>>>type(t3)
<class 'tuple'>
>>>type(t4)
<class 'int'>
>>>type(t5)
<class 'tuple'>
>>>type(t6)
<class 'int'>
>>>
```

还有一种特殊的元组就是空元组，如下：

```
>>>t7 = ()
>>>t8 = tuple()
>>>t7
()
>>>t8
()
```

tuple()的另一个作用就是将其他序列转为元组：

```
>>>t9 = tuple("hello")
>>>t9
('h', 'e', 'l', 'l', 'o')
>>>
```

5.7.2 元组赋值

元组除了可以把一组数据赋值给一个变量，还可以进行拆分赋值，比如这个经典的算法问题：有红墨水和蓝墨水各一杯，问怎样将杯中墨水交换？解决办法就是再找一个空杯做中转，用传统的赋值方式表示如下：

```
>>>red = "red water"
>>>blue = "blue water"
>>>temp = red
>>>red = blue
>>>blue = temp
```

这个办法在 Python 中就显得不那么简洁优雅了，在 Python 中，我们可以这样：

```
>>>red, blue = blue, red
```

这种等号左右都是元组的方式就是元组拆分，这种赋值方式需要注意的是左边变量数跟右边元组的元素数要相等。

5.7.3 列表和元组

列表和元组是可以相互转换的，各自都提供了转换函数 list()和 tuple()。除了相互转换之外，还有一种情况会把两者结合在一起。

Python 提供了一个内置函数 zip()，作用是接收多个序列，每个序列取一个值放到一个元组里，在 Python 2 中所有的元组组成一个列表，在 Python 3 中不能直接看到这个列表，zip()返回一个迭代器，需要迭代才能看到里面的值。如果序列长度不同，则以短的为准：

```
>>>x = 'xyz'
>>>l = [1,2,3,4]
>>>zip(x, l)
```

```
<zip object at 0x05450B48>
>>>for i in zip(x,l):
    print i

('x', 1)
('y', 2)
('z', 3)
```

迭代时，可以利用元组赋值方式直接拆分元组：

```
>>>for v, k in zip(x,l):
    print(k, "===>", v)

1 ===>x
2 ===>y
3 ===>z
```

这种组合用来解决某些问题是很有用的，比如判断两个序列当中是否在同一位置含有相同元素：

```
>>>for x, y in zip(l1, l2):
        if x == y:
            print(x)
```

5.7.4 什么时候使用元组

元组与同为序列的字符串和列表，应该根据什么决定是否使用元组呢？

首先是字符串，相比其他序列，字符串的限制比较多，元素只能是字符而且不可变。

列表是最灵活的，相比元组，用到的地方更多一些，所以本书用了较大篇幅介绍。

元组作为与列表相似的数据类型，重点在于它的不可变性，这种不可变性提供了一种具有完整性和持久性的数据结构。因此，可以为需要固定数据的地方提供不可变对象。比如后面会讲到字典类型的键就是不可变的，这时就只能用元组而不能用列表。

5.8 《英雄无敌》需求落地

现在我们研究一下游戏《英雄无敌》的需求适合用什么方式来实现。



《英雄无敌》迭代开发、构建英雄世界、优化数据结构、coding

(1) 开局玩家角色起名字：input。

(2) 列表初始化英雄属性：[名字，血值，攻击力，防御力]。

(3) 判断名字是否为空，默认为“玩家一”：if 语句。

(4) 判断英雄行动方向：if。

(5) 设计地图九宫格：列表或元组。

(6) 优化数据结构。

将字符串的《英雄无敌》做一下升级大致变成：

```
'''
Heroes beta-0.2
milo  str worldMap if while
'''
welcome = '- * - welcome to Heroes world!- * - '
mapmsg = '#######'
mapins = "\n- * - the world is like this - * - \n %s \n the '*' is you " % (mapmsg,)
worldMap = ['#','#','#','#','#','#','#']
instruction = '''
contrl your hero:| 'a'for left | 'd' for right |'''

print(welcome)

name = input('input your name:')
hp = 100

if not name:
    name = 'player01'

usermsg = {'name':name,'hp':hp}

print("HI!", usermsg['name'],'You Hp is :',usermsg['hp'])
print(mapins,instruction)

point = 0
while 1:
    worldMap[point] = "*"
    print('you are here',"".join(worldMap))
    userinput = input('go or quit:')

    if userinput == 'd' and point < 6:
        worldMap[point] = '#'
        point += 1
    elif userinput == 'a' and point > 0:
        worldMap[point] = '#'
        point -= 1
    elif userinput == 'quit':
        print("goodbey!!")
        break
    else:
        print(instruction)
```

当然，这只是一个思路，你设计的游戏是什么样子呢？自己实现吧！肯定比我这个有意思。

重点提示

在这一章中，你要掌握的内容：

(1) 熟练掌握列表和元组的各种操作。

(2) 会用列表推导式和生成器表达式。

(3) 理解深拷贝和浅拷贝。



动动手

(1) 由用户输入一系列值,输入 q 则结束,将所有的值保存为一个列表。

(2) 编写程序,实现对列表的倒序,如源列表为[1,2,3,4,5],倒序后为[5,4,3,2,1]。

(3) 编写一个篮球记分系统,操作者输入球队名字和获得分数,由程序进行累加并显示到屏幕上,然后等待下次输入球队得分,当输入 game over 时,屏幕显示比赛结束,并自动判断胜利一方,显示 Winer is ×××。

第 6 章 分治策略——函数与模块

分治策略基于“分而治之”的思想，在开发一个较大的程序时，最好的办法是基于较小程序组件来构建。较小的程序组件更灵活，程序更易于开发和维护。Python 中的程序组件包括函数、类、模块和包。本章先讲函数和模块。同时，这一章也可以给《英雄无敌》带来更多的可能，比如增加随机发生事件。

6.1 函数基础

函数就是一段具有特定功能、被封装、可重用的语句块，通常用来实现某一个特定的功能。给这段程序起一个名字，就可以在程序的任何地方通过这个名字任意次地运行这个语句块。这也是函数的两个重要概念：定义和调用。



函数：声明、变量作用域、多参数冗余处理

函数的工作常态就是接收数据，在内部进行处理，返回处理结果。一个有意思的比喻用来形容这样的模型叫黑箱模型，这是个比较形象的比喻，指一段封装了特定功能的程序，对于需要这种功能的用户来说，并不需要知道内部实现的原理和过程，程序本身提供了输入接口，经过内部处理后对用户返回结果。黑箱模型在编程领域非常常见，包括人工智能实际也是黑箱模型。

函数可以简化脚本，在前面的学习中，已经接触并运用了一些 Python 提供的内建函数。我们并不需要了解函数内部怎样实现的，只是拿来一个名字就可以实现很多看似复杂的功能，大大减少了程序代码的复杂度。

除了 Python 提供的内建函数，还可以根据需要编写自己的函数，通常用来完成大量重复或特定的功能，这跟循环不一样，循环是一次性反复执行一段代码，而函数可以在需要时随时被多次调用。

6.1.1 自定义函数

自定义函数通过 def 关键字定义。def 关键字后就是函数的标识符也就是函数名，函数提供的输入接口就是函数名后面的圆括号，圆括号中是变量名，在函数中称为参数，一个函数的参数数量视函数功能决定。圆括号后以冒号结尾，接下来是函数的语句块，函数语句块相对于 def 要缩进，具体格式如下：

```
def 函数名(参数):
```

```
    语句块(函数体)
```

我们先看一个无参数示例，把之前的九九乘法表放到函数里，这样就可以随时打印九九乘法表了，为控制篇幅，我们打印到 5：

```
#fun99.py
def fun99():                        #定义函数
    for i in range(1, 6):
        for j in range(1, i + 1):
            print(j, '*', i, '=', i * j, '\t', end='')
        print('\n')

print("It's Main")                  #主程序的其他代码,跟函数无关
fun99()                             #调用函数
```

这个例子定义了一个名字叫 fun99 的函数，括号中为空的意思是不需要参数，最终的功能就是函数体的代码功能。调用函数的方法是在需要执行的地方通过函数名加圆括号就可以了。运行效果如下：

```
It's Main
1 * 1 = 1
1 * 2 = 2     2 * 2 = 4
1 * 3 = 3     2 * 3 = 6     3 * 3 = 9
1 * 4 = 4     2 * 4 = 8     3 * 4 = 12    4 * 4 = 16
1 * 5 = 5     2 * 5 = 10    3 * 5 = 15    4 * 5 = 20    5 * 5 = 25
```

6.1.2 形参和实参

自定义函数作为一个工具可能会在不同的情况下调用，而且可能需要根据情况给出不同的结果，比如打印 3×3 乘法表或打印 100×100 的乘法表就需要参数，参数作为函数的输入，由调用者决定传入函数不同的值，经过函数处理后返回对应的结果。

定义函数时，在函数名后圆括号内的参数叫形参，如果有多个形参，需要通过逗号隔开，这个形参并不是具体的值，它相当于一个变量名。

调用函数时可以通过参数给函数传值，通过参数赋值的过程叫传参。而调用函数时这个参数就叫实参，通常为了交流方便，并不会特意说形参或实参，而直接说参数。

现在我们先改造一下乘法表的函数，通过参数控制打印的具体值：

```
#fun99_1.py
def fun99(x):                           #设置一个形参
    for i in range(1, x + 1):           #实参的值可以传递到函数体内
        for j in range(1, i + 1):
            print(j, '*', i, '=', i * j, '\t', end='')
        print('\n')

print("It's Main")
fun99(3)                                #数字 3 即是实参的值
fun99()                                 #调用函数未传参
```

上述代码设置了一个形参 x，当调用这个函数时就需要给 x 传值，这个值会传递到函数内部，有参数函数调用时需要注意的是实参的数量要与形参相同，不传参或多传参都会产生异常，运行效果如下：

```
>>>
It's Main
1 * 1 = 1
1 * 2 = 2     2 * 2 = 4
1 * 3 = 3     2 * 3 = 6     3 * 3 = 9
Traceback (most recent call last):
  File
"C:/Users/milo/Desktop/CrazyPythonZeroToHero/code/fun99_1.py", line 10, in <module>
    fun99()
TypeError: fun99() missing 1 required positional argument: 'x'
>>>
```

关于参数的类型我们会在第 6.3 节介绍。

6.1.3　返回值

上面的例子中，我们通过 print()直观地看到函数的执行效果，但在使用过的内建函数中，大部分都会返回一个结果，而不是打印到屏幕上。这些返回的结果就是函数的返回值，它可以赋值给一个变量，或者作为其他表达式的一部分。

```
>>>max(1,2,3)
3
>>>x = max(1,2,3)
>>>print(x)
3
```

返回值是函数的一部分，即使不设置，依然也有返回值。例如：

```
x = fun99(3)      #等号后调用了函数，此时就已经向屏幕打印结果了，函数的返回值赋值给了 x
print(x)          #打印函数的返回值
```

效果如下：

```
It's Main
1 * 1 = 1
1 * 2 = 2     2 * 2 = 4
1 * 3 = 3     2 * 3 = 6     3 * 3 = 9
None
```

因为没有设置返回值，所以看到的是 None。

自定义返回值用 return 语句。return 语句后面是一个表达式，这个表达式可以很复杂也可以是一个值。调用函数相当于进入函数，执行到 return 语句从函数中返回，同时表达式的值作为返回值。

下面是一个能够返回给定半径的圆的面积的函数：

```
def area(radius):
    return 3.14 * radius* * 2
```

每个函数只会有一个返回值，如果有多个 return 语句，只会执行第一个。但是我们可以设计一些分支来返回不同的返回值，比如下面这个获得绝对值的函数：

```
#absolute.py
def absolute(x):
    if x < 0:
        return - x
    else:
        return x
    print(x)        #无效代码
```

需要注意的是，return 语句一旦执行，函数就会终止，其后的其他语句都不会执行。

另外，Python 中提供了一个内建函数 abs()，用来计算绝对值。这里要说的是程序员中的一个共识：不要重复造轮子。简单地说就是已经存在的功能可以直接拿来用，练习时可以尝试写一些，工作中不要什么都自己写，费时费力还不高效。

6.2 变量作用域

变量作用域指的是变量起作用的范围，涉及函数编程就会有变量作用域的问题。比如我们在函数内部定义的变量和函数外部定义的变量，作用域是有区别的。

6.2.1 局部变量

局部变量是在函数内定义的变量，这种变量只能在函数内部使用。局部变量的作用域只在它被定义的语句块中。

下面的例子用来说明局部变量：

```
#func_var1.py
x = 10
def func(i):
    print("x = ", i)
    x = 100
    y = 50
    print("local x = ", x)
    print("local y = ", y)

func(x)
print("x = ", x)
print("y = ", y)
```

在这个例子中，主程序中定义了变量 x 的值为 10，定义函数时，在函数中也定义了一个变量 x，值是 100，当然，这两个变量没有任何关系，另外我们还定义了一个变量 y。当调用函数时将主程序的 x 的值传给函数，函数内打印过一次之后，将 x 的值重新定义为 100，之

后再打印一次。当函数调用完毕后再打印一次主程序中 x 的值。可以看到主程序的变量并没有被函数内的赋值影响。而 y 的值只在函数内可以被调用，在函数之外则不存在。运行结果如下：

```
x = 10
local x = 100
local y = 50
x = 10
Traceback (most recent call last):
  File
"C:/Users/milo/Desktop/CrazyPythonZeroToHero/code/func_var1.py", line 13, in <module>
    print("y = ", y)
NameError: name 'y' is not defined
```

6.2.2 全局变量

如果想要函数内的变量作用于函数之外，这种变量的作用域就必须是全局的。能够作用于函数内外的变量，叫作全局变量。

全局变量可以通过 global 语句定义，改造一下上面的程序，把 x 和 y 定义成全局变量：

```
#func_var2.py
x = 10
def func(i):
    global x, y
    print("x = ", i)
    x = 100
    y = 50
    print("local x = ", x)
    print("local y = ", y)

func(x)
print("x = ", x)
print("y = ", y)
```

运行结果如下：

```
x = 10
local x = 100
local y = 50
x = 100
y = 50
```

这个例子中，主程序定义过的 x 在函数内部声明了全局变量，调用了函数之后，x 的值也跟着变了。主程序没定义过的 y 在函数内部声明为全局变量，在函数调用之后，也可以在主程序中调用了。

6.2.3 命名空间

Python 中有严格的变量作用域的区分，主要依据就是变量定义的位置，也就是命名空

间。Python 的命名空间共有三个：局部、全局和内建。程序访问变量时会按局部、全局、内建的顺序搜索命名空间。

比如我们在一个函数中访问一个变量时，就会先看这个函数中是否有这个变量，如果没有就会再搜索全局，如果还没有就会搜索内建命名空间。不同函数拥有独立的命名空间，即不同函数中名字相同的变量并不会相互影响。

了解了命名空间，还有一个问题需要注意，就是尽量避免同名的全局变量、局部变量以及函数的参数定义。因为很容易相互影响，而且代码也不易阅读。

6.3 参数的类型

在第 6.1 节中，我们知道传参可能会遇到的问题，有参数不传参、多传或少传等都会产生异常，这一节我们来看看怎么解决这些问题。

6.3.1 默认参数

函数设置了形参后，如果不传参就会产生异常，可以通过给参数设置默认值来解决这个问题。参数设置了默认值后，在调用函数时如果不传参，就会使用默认值；如果有实参，则将实参重新传递给形参。设置默认参数的方法就是在定义函数时，通过等号为形参直接赋值。

下面是一个小例子：

```
#func_machine.py
def machine(money = 18, food = "套餐"):
    print("一份 %d 元 %s" % (money, food))

machine()
machine(36)
```

这个例子中的两个参数都设置了默认值，调用时如果不传参，则使用默认值；如果实参传值，则用新值。运行效果如下：

```
一份 18 元套餐
一份 36 元套餐
```

如果有多个形参，就需要特别注意，因为函数调用时，参数赋值默认是从左至右依次赋值的。所以，如果多个形参中有默认参数，要从右至左设置，不可以先声明默认参数，在右侧再出现没有默认值的形参，比如下面几种形式：

```
def machine(money, food = "套餐")              #正确
def machine(money = 18, food)                  #错误
def machine(money, food = "套餐", other)       #错误
```

6.3.2 关键参数

如果函数中有多个参数，可以通过参数名字作为关键字给参数赋值，这时可以不按顺



序赋值。例如：

```
#func_machine_x.py
def machine(money, food = "套餐", other = ""):
    if not other:
        print("一份 %d 元 %s" % (money, food))
    else:
        print("一份 %d 元 %s 外加 %s" % (money, food, other))
machine(money = 30, other = "可乐")
```

在这个例子中，我们跳过 food 指定了 money 和 other 的值，效果如下：

```
一份 30 元 套餐 外加 可乐
```

关键参数的好处就是可以不按顺序传值。

6.3.3 冗余参数处理

传值少可以通过默认值解决，但有的时候某些函数被调用时，传递参数的个数比形参多，比如这样调用上面例子的函数：

```
machine(100, "比萨", "可乐", "橙汁", "苏打水")
```

就会看到这样的异常提示：

```
TypeError: machine() takes from 1 to 3 positional arguments but 5 were given
```

异常提示说得很清楚，这个函数只需要 1～3 个值，但是给了 5 个。这种情况的解决办法就是定义函数时设置可变长的参数。通过在参数前加“*”即可实现。

```
#func_machine_list.py
def machine(money, food = "套餐", *other):
    print(other)
    if not other:
        print("一份 %d 元 %s" % (money, food))
    else:
        print("一份 %d 元 %s 外加 %s" % (money, food, other))
machine(100, "比萨", "可乐", "橙汁", "苏打水")
```

上例在 other 前添加了“*”标识符，因为传参顺序的原因，混合普通参数和可变长度参数时，可变长度参数放在最右边。效果如下：

```
('可乐', '橙汁', '苏打水')
一份 100 元比萨外加 ('可乐', '橙汁', '苏打水')
```

可以看到，有了可变长度参数之后，多余的值被以元组的形式保存起来。

另外，对于关键字传值，需要给形参加“**”标识符，例如：

```
def func(**args):
```

```
    print(args)

func(x = 1, y = 2, z = 3)
```

运行效果如下：

```
{'x': 1, 'y': 2, 'z': 3}
```

这时，多余的参数被以字典的形式保存下来。

有了这么多的参数处理形式，在实际项目中就可以灵活运用来实现不同的功能了。

6.3.4 序列和字典做实参

如果将序列和字典作为形参传递给函数会怎么样呢？先看一个小例子：

```
li = [1, 2, 3]
di = {'x':1, 'y':2,'z':3}
def func(x):
    print(x)

func(li)
func(di)
```

结果如下：

```
[1, 2, 3]
{'x': 1, 'y': 2, 'z': 3}
```

很明显，列表也好，字典也好，都是作为一个整体传给了函数，如果像下面这个函数正好需要 3 个值，可以在调用函数通过实参传值时把它们分解开来，方法是通过加“*”：

```
#func_machine_list_dict.py
li = [1, 2, 3]
di = {'x':1, 'y':2,'z':3}
def func(x, y, z):
    return x*y*z

func(*li)
func(**di)
```

需要注意的是，拆分列表用“*”，拆分字典用“**”，并且字典的 key 要跟函数的形参一致。

6.4 内建函数

Python 中提供了大量的内建函数，在功能实现之前可以先看看是否已经存在相关函数，比如 len()、max()、abs()等，手册有详细说明（见图 6-1）。

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

图 6-1　Python 3 内建函数

下面我们了解几个有意思的内建函数。



内建函数：不要自己造轮子

1. filter

filter(function,iterable)：筛选过滤，循环可迭代的对象，把迭代的对象当作函数的参数，如果符合条件，返回 True；否则，返回 False。

例如，过滤掉一个列表中的所有偶数：

```
#func_filter.py
li = [1, 2, 3, 5, 7, 8, 11, 18]
def func(x):
    if x%2 != 0:
        return True

ret = filter(func, li)
for i in ret:
    print(i)
```

运行效果如下：

```
1
3
5
7
11
```

filter()函数有个特别的地方就是第一个参数必须是一个函数，刚开始接触会有些不适应，具体应用时只要把手册说明看明白，按手册的要求用就可以了。

2. map

map(function,iterable,...)：将序列中的每一个元素都传到函数中执行并返回，可以同时遍历多个序列，如果长短不一，以短序列为准。

例如，将两个序列的元素分别相加：

```
#func_map.py
l1 = [1, 2, 3]
l2 = [5, 7, 8, 11]
def func(x, y):
    return x + y
for i in (map(func, l1, l2)):
    print(i)
```

map()也用到了函数参数，结果如下：

```
6
9
11
```

因为取短的原因，所以，虽然第二个序列有 4 个值，但结果只有 3 个。

6.5 匿名函数：lambda 表达式

lmabda 表达式的作用是实现一个轻便的函数功能，又不需要起名字，所以也叫匿名函数。


匿名函数 lambda

什么时候会用到函数却不需要名字呢？如果观察 6.4 节的两个例子就会发现，其中的函数参数都是为 filter()和 map()定制的，也只有在 filter()和 map()中才有用，这种函数就可以用 lambda 表达式来代替。

lambda 表达式的语法为

```
lambda 参数：返回值表达式
```

前面用于 filter()的函数可以写成：

```
lambda x:x%2 != 0
```

这只是一个表达式，并不能被调用，因为没有名字。如果想调用，可以起个名字，不过，这样做的意义不大，也违背了 lambda 表达式的初衷。例如：

```
func = lambda x:x%2 != 0
func(li)
```

6.4 节 filter()的例子就可以写成：

```
>>>filter(lambda x: x%2 != 0,li)
```

6.4 节 map()的例子可更改如下：

```
>>>map(lambda x,y: x + y, l1, l2)
```

有了匿名函数就可以使程序更简洁,但并不是不分情况地乱用,比如这两个例子还有一个特点,都是通过对序列遍历生成新的序列或迭代对象,这个特点正是列表解析和生成器表达式可以做的事。所以,这两个例子还可以写成列表解析的形式。

用列表解析改写 filter()的例子如下:

```
>>>li = [1, 2, 3, 5, 7, 8, 11, 18]
>>>[i for i in li if i%2 != 0]
[1, 3, 5, 7, 11]
```

用列表解析改写 map()的例子如下:

```
>>>l1 = [1, 2, 3]
>>>l2 = [5, 7, 8, 11]
>>>[x + y for x,y in zip(l1, l2)]
[6, 9, 11]
```

这里用了一个函数 zip(),它有什么用? 请你查手册自学一下吧。

虽然这几个例子可以互相转化,但只是因为这几个方法正好都可以用来解决问题而已,并不是 lambda 跟列表解析存在必然联系。在实际设计程序时,根据上下文,选择合适的方法即可。

6.6 生成器 yield 语句

函数可以通过 return 返回一个值,但是通过 yield 语句却可以让函数分多次返回多个值。

简单地说,生成器就是包含 yield 关键字的函数。本质上来说,关键字 yield 是一个语法糖,内部实现支持了迭代器协议。同时,yield 内部是一个状态机,维护着挂起和继续的状态。


生成器 yield

小知识:

语法糖(syntactic sugar):指计算机语言中添加的某种语法,这种语法对语言的功能并没有影响,但是更方便程序员使用。通常使用语法糖能够增加程序的可读性,从而减少程序代码出错的机会。不过它并没有给语言添加任何新东西。

那么,生成器是怎么调用执行的呢? 只需要了解下面几条规则即可。

(1) 当生成器被调用时,函数体的代码不会被执行,而是返回一个迭代器,即生成器函数返回的是生成器的迭代器。“生成器的迭代器”这个术语通常被称作“生成器”。要注意的是生成器就是一类特殊的迭代器。作为一个迭代器,生成器必须要定义一些方法,其中一个就是 next()。如同迭代器一样,我们可以使用 next()函数来获取下一个值。这一切都是在 yield 内部实现的。

(2) 当 next()方法第一次被调用时,生成器函数才开始执行,执行到 yield 语句处停止。next()方法的返回值就是 yield 语句处的参数(yielded value)。当继续调用 next()方法时,函数将在上一次停止的 yield 语句处继续执行,并到下一个 yield 处停止;如果后面没有 yield 就抛出 StopIteration 异常。

(3) 每调用一次生成器的 next()方法,就会执行生成器中的代码,直到遇到一个 yield

或者 return 语句。yield 语句意味着应该生成一个值(在上面已经解释清楚)。return 意味着生成器要停止执行,不再产生任何东西。

(4) 生成器的编写方法和函数定义类似,只是在 return 的地方改为 yield。生成器中可以有多个 yield。当生成器遇到一个 yield 时,会暂停运行生成器,返回 yield 后面的值。当再次调用生成器时,会从刚才暂停的地方继续运行,直到下一个 yield。生成器自身又构成一个循环器,每次循环使用一个 yield 返回的值。

一个比较直接的例子如下:

```
#func_yield.py
def f():
    print("one")
    yield 1
    print("two")
    yield 2
    print("three")

g = f()
gy = next(g)
print(gy)
gy = next(g)
print(gy)
gy = next(g)
print(gy)
```

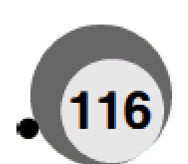
运行结果如下:

```
one
1
two
2
three
Traceback (most recent call last):
  File
"C:/Users/milo/Desktop/CrazyPythonZeroToHero/code/func_yield.py", line 15, in <module>
    gy = next(g)
StopIteration
```

从结果可以直观地看到效果,含有 yield 语句的函数被调用后就会返回一个生成器对象,前两次通过 next()函数都可以取到 yield 返回的值,第三次 next()函数在执行完函数体代码后因为没有 yield 了,所以抛出了 StopIteration 异常。

6.7 模块和包

通过函数我们可以把大问题分解为若干小问题,并且可以让程序更灵活,不过有时在一个项目中的一些函数或类在几个程序中都会用到,这时就要通过模块或者包来达到代码重用的目的。



6.7.1　模块

Python 模块(module)其实是一个 Python 文件,以 .py 结尾,包含了 Python 对象定义和 Python 语句。换句话说,其实每个 Python 脚本都可以被当作模块。



模块化开发:
自定义模块

模块化编程的好处是能让你有逻辑地组织 Python 代码段,并且把相关代码分配到一个模块里,能让代码更好用、更易懂。在模块中能定义函数、类和变量,模块里也能包含可执行的代码。

下面的脚本就可以被当作一个模块:

```
#support.py
def myAdd(x):
    return x + 100
```

6.7.2　导入模块

导入模块的方式是通过 import 语句或 from...import 语句,在一个程序中每个模块只需要导入一次。import 语句的语法为

```
import module1[, module2[,... moduleN]
```

要调用模块中的方法,在模块导入后,需要用"模块名.函数名"的方式。

在之前的演示中已经用过 import 语句很多次,现在要导入自己写的模块。可以这样导入前面的 support.py 模块,并使用里面的方法:

```
#test.py
import support
support.myAdd(5)
```

第一次导入自己写的模块时需要注意,当解释器遇到 import 语句,如果模块在当前的搜索路径,会被导入;否则,会失败。搜索路径是指一个解释器会先进行搜索的所有目录的列表。如果你还不清楚搜索路径,可以将 support.py 和 test.py 放到同一个目录下。

如果一个模块中有很多函数,但是只需要用到一个或者几个,就需要用 from 语句了,语法如下:

```
from modname import name1[,name2[,... nameN]]
```

比如,导入 support 模块的 myAdd 函数,并调用:

```
from support import myAdd
myAdd(5)
```

通过这种方式导入的函数可以直接调用,可以通过下面的语句导入所有函数:

```
from support import *
```

虽然这样导入后，在调用函数时会方便些，但并不建议，因为很有可能跟当前程序中的函数弄混。

最后一个小技巧就是如果导入的模块名字比较长，可以通过取别名的方式简化。名字可以自己随便起，一些比较常用的模块都有公认的别名，比如 numpy 通常起名叫 np，这样其他人在看代码时也第一时间就能反应过来你程序中的别名代表什么。

```
>>>import numpy as np
```

6.7.3 搜索路径

Python 在导入模块时，会按顺序自动搜索模块，搜索过程如下。

(1) 当前目录。

(2) 当前目录没有，则搜索 shell 变量 PYTHONPATH 下的所有目录。

(3) 如果都找不到，Python 会查看默认路径。UNIX/Linux 下，默认路径一般为/usr/local/lib/python/；Windows 通常在 C:\python27\lib(取决于你的安装路径)。

6.7.4 包


模块化开发：包

模块通常是包含了很多函数或类的一个脚本，而包则可以理解为是包含了很多模块的一个目录(文件夹)。

包是一个分层次的文件目录结构，它定义了一个由模块及子包和子包下的子包等组成的 Python 的应用环境。

包就是一个文件夹，但该文件夹下必须存在 __init__.py 文件，用于标识当前文件夹是一个包，该文件的内容可以为空。你可能会发现，没有__int__.py，包一样可以用，但是当其他人看你的代码结构时，如果没有__int__.py，其他人是不太可能第一时间就知道这目录是个包。

现在我们把前面的 support.py 放到文件夹 milo 里，并在这个 milo 文件夹里建一个空文件__init__.py。这时的 milo 就是一个包了。

test.py 是用来测试包的脚本。

导入模块跟导入包类似，只是结构上多了一级，在导入时注意包的结构就可以了，比如下面这些方法都是可以的：

```
#test.py
import milo.support
milo.support.myAdd(5)

#test.py
from milo import support
support.myAdd(5)

#test.py
from milo.support import myAdd
myAdd(5)
```

6.7.5　__name__属性

Python 内置的__name__属性可以用来识别程序是被导入的还是直接执行的。因为文件如果是以顶层程序文件(主程序)执行的,__name__值为__main__,如果文件是被导入的,__name__的值就是当前脚本的名字。

如果编写的模块中有需要执行的程序,而这些程序我们并不想让调用者执行,那就需要__name__了。原理很简单,先看下__name__在两种情况下的值。

```
#support.py
def myAdd(x):
    return x + 100

print(__name__)
```

```
#test.py
import supprt
```

分别运行 support.py 和 test.py,运行结果如下:

```
>>>__main__
```

```
>>>support.py
```

可以看到,在模块中,我们打印了__name__的值,直接运行的结果是__main__,但是在 test.py 中,我们只是导入了模块,也打印了一个字符串,就是模块文件的名字。这是因为,我们在导入模块时,实际上就是加载了一次模块当中的代码,所以,模块中的 print()函数被执行,但此时的__name__值就变成了模块文件的名字。根据这个功能,可以避免模块中的执行语句被导入执行了:

```
#support.py
def myAdd(x):
    return x + 100

if __name__ == '__main__':
    print("作为主程序运行")
else:      #else 不是必需的
    print("导入 %s 模块" % (__name__,))
```

if __name__ == '__main__':的用法是非常常用的,特别是脚本中有可被调用的函数或类时,即便没人调用,通常习惯上也会这样执行主程序。

重点提示

在这一章中,你要掌握的内容:

(1) 会使用自定义函数解决问题。

(2) 明确变量作用域。

(3) 明确参数和返回值的意义和用法。

(4) 会使用 lambda 表达式和 yield 语句。

(5) 明确饱和模块的意义以及调用方法。



动动手

(1) 编写一个函数,判断用户输入的三个数字是否构成一个三角形。

(2) 编写一个函数,接收传入的字符串,分别统计大写字母、小写字母、数字以及其他字符的个数,以元组的方式返回这些数字。

(3) 通过 help()或手册学习内建函数 random()的作用,并且尝试写一个生成验证码的函数,比如每次从自定义字符集中随机产生一个 4 位的验证码。

(4) 函数有一个特征,就是返回值,这个返回值甚至可以返回一个函数,希望你在书本之外可以多了解一下装饰器和闭包这两个概念,这对基础概念的理解帮助不大,但有时可以提高编程效率,我准备了视频,去看看吧。



装饰器



闭包

(5) 做运维工作经常会分析各种各样的日志文件,Apache 是一个非常著名的 Web 服务器,它的日志文件特点是每行记录以空格来分割不同的记录,如果你在工作中有接触,尝试以空格为分割符来解析 Apache 日志文件,这里有一个视频,你可以了解一下。

(6) 重新设计《英雄无敌》游戏,通过函数实现角色在行进过程中踩到陷阱、加血等事件,并对角色的血量有增加或减少的影响,如果没有头绪,看看视频参考吧。



解析 Apache 日志:
空格分割



《英雄无敌》:路途
坎坷及随机事件

第 7 章 字典和集合

为了涵盖现实世界的各种信息，我们已经学习了数字、字符串、列表和元组这些数据类型。特别是掌握了列表类型后，我们可以对数据做很灵活的操作，就好像在程序运行过程中有个数据库一样。字典和集合也是 Python 中提供的数据类型，字典的功能更是非常强大，除了作为一种数据结构甚至还可以变通为一种语法结构，这也使字典在解决很多问题时都非常有用。

7.1 字典

字典是类似于列表的一种数据结构，不同的是，字典是键值对的无序集合。字典中的每个元素由两部分组成：键（key）和值（value），每个键都映射到一个值上。序列的索引是 0 起始的数字，字典的索引就是键，通过键访问值。


字典：创建和基本操作

字典强大的地方在于，字典的键可以是任何不可变对象，比如数字、字符串、元组等，值可以是任意类型，比如数字、字符串，甚至一个函数。

7.1.1 创建字典

定义字典的方式是通过花括号{}将所有元素括起来，其中的元素用冒号分隔键和值，像下面这样，创建一个 hero 对象：

```
>>>hero = {'id':1, 'name':'milo', 'hp':100}
>>>hero
{'id': 1, 'name': 'milo', 'hp': 100}
>>>type(hero)
<class 'dict'>
```

需要注意的是，不要用 dict 做变量名，因为 dict 是内建函数的名字，我们可以通过 dict() 创建一个字典，比如：

```
>>>dd = dict()
>>>dd
{}
```

这样就有了一个空字典,但通常我们都会直接用花括号创建一个字典。

还有一种方法就是通过 zip()创建一个字典,zip()可以从两个序列中创建成对的元素,再通过 dict()函数就可以得到一个字典。

```
>>>keys = ['id', 'name', 'hp']
>>>values = [1, 'milo', 100]
>>>dict(zip(keys, values))
{'id': 1, 'name': 'milo', 'hp': 100}
```

7.1.2 字典的键和值

序列中我们通过 0 起始的索引访问元素,而字典是无序集合,所以不存在顺序索引。在字典中的索引就是键,只能通过键访问值,也不能反过来,这一点就好像你查字典可以根据一个字找到它的解释,但不能反过来通过解释找到字。所以需要格外注意键和值的设置。

字典的元素同列表一样是可变的,我们可以对键赋值,除了直接赋值,字典也提供了丰富的方法,在后面会介绍。

字典类型有两个优势:一个是索引速度更快;另一个是字典的值可以是任何对象,更主要的是你可以给每个值都起个名字。

```
>>>demo = {'d':{'hp':100, 2:2.2},
          3:[100, 200, 300],
          (1,2,3):6}
>>>demo[3]
[100, 200, 300]
>>>demo[(1,2,3)]
6
>>>demo['d']
{'hp': 100, 2: 2.2}
>>>demo['d'][2]
2.2
>>>demo[3] = '333'
>>>demo['d']['hp'] = 90
>>>demo
{'d': {'hp': 90, 2: 2.2}, 3: '333', (1, 2, 3): 6}
```

这个字典中包含了几种不同的键和值,定义字典时,如果元素比较多,可以分多行定义,这样阅读起来比较方便。取值的时候要注意键的类型以及所对应的值的类型,比如['d']就又生成了一个字典,所以,可以再通过新字典的键进行操作。

7.1.3 字典的相关操作

之前介绍了很多针对数据集合的操作,对字典同样适用。需要注意的是,字典是可变的,并且在字典中的键值映射中,键起了关键作用。

len():统计字典中元素的个数,即有多少键值对。

in:成员测试,用来测试键是否在字典中。



for：遍历字典，通过迭代字典的键实现循环。

下面集中演示几个例子，注意例子中所操作的对象是键还是值：

```
>>>demo = {'milo':'123456', 'zou':'654321', 'qixian':'987654'}
>>>demo
{'milo': '123456', 'zou': '654321', 'qixian': '987654'}
>>>len(demo)
3
>>>'milo' in demo
True
>>>'123456' in demo
False
>>>for x in demo:
    print(x,demo[x])

milo 123456
zou 654321
qixian 987654
```

7.1.4　字典的方法

字典类型中也提供了很多方法，通过 help(dict)就能看到，有一些比较常用的，我们来看一下。

7.1.3 小节我们遍历字典时只能取到键，通过 values()方法就可以获取一个包含所有值的列表，当然通过 keys()获取的就是所有的键，你也可以通过 items()一次性获取所有键值对，items()返回的是一个由元组形式返回的键值对组成的列表。有了这几个方法，我们在遍历字典的时候就方便多了。

```
>>>demo = {'milo':'123456', 'zou':'654321', 'qixian':'987654'}
>>> >>>demo.values()
dict_values(['123456', '654321', '987654'])
>>>for value in demo.values():
    print(value)

123456
654321
987654
>>>demo.keys()
dict_keys(['milo', 'zou', 'qixian'])
>>>demo.items()
dict_items([('milo', '123456'), ('zou', '654321'), ('qixian', '987654')])
>>>for key, value in demo.items():
    print(key, '==>', 'value')

milo ==> value
zou ==> value
qixian ==> value
```

7.2 字典实例：统计高频词

通过字典，可以做很多事情，比如分析一首老歌 *Killing Me Softly With His Song* 中出现频率最高的单词。

要实现这个功能其实有很多方法，除了使用字典之外，根据前面学习过的知识，也能找出几种解决方法。比如通过变量列表的方式等。如果使用字典，这个过程就会更加简单明了。可以创建一个空字典，然后遍历数据中的单词，以单词作键，计数器作对应的值。每遍历一个单词时，需要判断这个单词是否已经在字典中。如果存在，则计数器加 1；如果不存在，则创建这个键，计数器记为 1。把这个核心功能做成一个函数，只需要把数据传给它，由它来返回一个字典。KMSWHS. txt 是歌词文件，跟脚本放在同一目录。

```
#count_words_dict_2.py
contents = open('KMSWHS.txt','r').read() #读取全部歌词
cList = contents.split()

def histogram(contents):
    d = {}
    for w in contents:
        if w in d:
            d[w] += 1
        else:
            d[w] = 1
    return d

h = histogram(cList)
print(h)
```

这里用到的 open()作用是打开文本文件，open()对象的 read()方法则是一次性读取文件内的所有数据。运行结果如下：

```
{'killing': 19, 'me': 22, 'softly': 18, 'with': 35, 'his': 34, 'song': 19, '-': 1,
'roberta': 1, 'flack': 1, 'strumming': 6, 'my': 20, 'pain': 6, 'fingers': 5, 'singing':
8, 'life': 12, 'words': 11, 'telling': 6, 'whole': 6, 'i': 8, 'heard': 2, 'he': 12,
'sang': 2, 'a': 4, 'good': 1, 'had': 1, 'style': 1, 'and': 6, 'so': 1, 'came': 1, 'to':
3, 'see': 1, 'him': 1, 'listen': 1, 'for': 1, 'while': 1, 'there': 2, 'was': 3, 'this':
1, 'young': 1, 'boy': 1, 'stranger': 1, 'eyes': 1, 'felt': 2, 'flushed': 1, 'fever':
1, 'embarrassed': 1, 'by': 1, 'the': 1, 'crowd': 1, 'found': 1, 'letters': 1, 'read':
1, 'each': 1, 'one': 1, 'out': 1, 'loud': 1, 'prayed': 1, 'that': 1, 'would': 1,
'finish': 1, 'but': 1, 'just': 2, 'kept': 2, 'right': 2, 'on': 2, 'as': 2, 'if': 2,
'knew': 1, 'in': 1, 'all': 1, 'dark': 1, 'despair': 1, 'then': 1, 'looked': 1,
'through': 1, "wasn't": 1, 'clear': 1, 'strong': 1, 'oh': 2, 'la': 3, 'yeah': 1}
```

仔细观察发现很多单词只出现了一次，可以过滤掉它们，或者 10 次以下的单词都过滤掉，最后再做一个漂亮的输出效果：

```
def show(wordsDict):
    dictList = []
    for key, val in wordsDict.items():
        dictList.append((val, key))
    dictList.sort(reverse = True)          #倒序排序
    print('%-10s%10s' % ('word', 'count'))
    print('-' * 25)
    for val,key in dictList:
        if val > 10:
            print('%-12s           %3d' % (key, val))

h = histogram(cList)
show(h)
```

结果如图 7-1 所示。

```
word           count
-------------------------
with             35
his              34
me               22
my               20
song             19
killing          19
softly           18
life             12
he               12
words            11
```

图 7-1 运行结果

从结果看很有意思，出现频率最多的是歌名里的几个单词，也可以看出作者要表达的重点。

7.3 字典的妙用

字典的灵活还给我们带来了一种更高效的用法。比如前面讲的计算器程序，需要编写加、减、乘、除等好多方法，再通过多个 if 语句判断用户要做什么，然后调用相应的函数；《英雄无敌》游戏中判断玩家的行进方向至少有两个方向，标配应该是四个，如果都用 if 判断就会很低效，下面看一下如何用字典解决。



字典的妙用：语法结构

```
#dict_get.py
a = 1
b = 2
c = input('operator>')

def myAdd(x, y):
    return x + y
```

```
def mySubtract(x, y):
    return x - y

result = {  '+' : myAdd(a,b),
            '-' : mySubtract(a,b)
        }

print(result.get(c, "just '+ 、- '"))
```

上述程序的做法是把函数作为字典元素的值，再通过字典的 get()方法取值，get()会接收一个键和一个默认值，如果键在字典中，则返回对应的值，否则返回默认值。

7.4 集合

在 Python 中的集合是一组对象的集合，集合中的任何元素都不重复，元素没有顺序，所以集合跟字典一样也不是序列。



集合

7.4.1 Python 集合

创建集合的方式跟其他数据结构不同，没有符号化的快捷方式，只能通过 set()函数创建，set()函数可以没有参数，这样创建的就是一个空集。如果有参数，则只能有一个参数，而且这个参数必须是可迭代的，比如字符串、列表等，生成集合后是没有重复元素的。

```
>>>nullSet = set()
>>>nullSet
set()
>>>xSet = set('aabbccdd')
>>>xSet
{'c', 'a', 'd', 'b'}
>>>ySet = set([1,3,5,7,3,5])
>>>ySet
{1, 3, 5, 7}
>>>print(ySet)
{1, 3, 5, 7}
>>>zSet = set([1, (2,3), 4.5])
>>>zSet
{1, (2, 3), 4.5}
>>>for i in xSet:
    print(i)

c
a
d
b
>>>'a' in xSet
True
```

```
>>>len(xSet)
4
```

从返回值可以看出，空集显示的是 set()，其他的则用的是{}，跟字典不同的是只有值，没有键，所以不能通过索引进行取值。在 Python 2 中所有的集合都会显示 set()。

集合的元素可以通过 for 遍历，可以用 len()函数统计长度，也可以用 in 判断集合中是否含有某个元素，这些跟其他集合类型的数据相同。

7.4.2 集合的方法和应用

Python 集合不能通过索引操作，集合的所有操作都是通过方法，可以通过 help(set)看到所有集合提供的方法。

除了集合元素的添加、删除之外，交集、差集等这些操作也是通过对应的方法实现的。通常的操作方法是一个集合调用方法，另一个集合作为方法的参数，有些方法需要注意哪个做参数，哪个做调用者。

下面我们来看各种集合的实现方法，顺便再复习一下数学知识。先来创建三个集合，在后面的例子中使用。

```
>>>xSet = set("1234")
>>>ySet = set("3456")
>>>zSet = set("123456")
>>>xSet
{'2', '1', '4', '3'}
>>>ySet
{'3', '4', '6', '5'}
>>>zSet
{'3', '6', '1', '5', '2', '4'}
```

1. 交集

交集是两个集合中重叠的元素组成的集合，通过 intersection()方法创建交集。

```
>>>xSet.intersection(ySet)
{'3', '4'}
```

2. 并集

并集包含两个集合的所有元素，通过 union()方法创建。

```
>>>xSet.union(ySet)
{'3', '6', '1', '5', '2', '4'}
```

3. 差集

差集是保留不在另一个集合中的元素，类似做减法，所以要注意集合的调用关系。通

过 difference()方法实现差集,difference()方法的作用是收集存在于调用集合但不存在于参数集合的元素。

```
>>>xSet.difference(ySet)
{'2', '1'}
>>>ySet.difference(xSet)
{'6', '5'}
```

4. 对称差

对称差是与交集相反的集合,保留两个集合不同的元素,相当于并集减合集,通过 symmetric_difference 实现。

```
>>>xSet.symmetric_difference(ySet)
{'6', '1', '5', '2'}
```

5. 判断子集和超集

判断两个集合的子集和超集关系会返回布尔值,而不是一个具体的集合。当集合 A 的每个元素都是集合 B 中的元素,则 A 是 B 的子集(subset),反过来 B 就是 A 的超集(superset),通过 issubset()和 issuperset()来判断。

```
>>>xSet.issubset(zSet)
True
>>>zSet.issubset(xSet)
False
>>>zSet.issuperset(xSet)
True
>>>xSet.issubset(xSet)
True
>>>xSet.issuperset(xSet)
True
```

6. 集合元素的增删方法

集合元素增删的方法如下。

add():向集合添加元素,如果元素已经存在,则不会有变化。

remove()和 discard():删除集合中的元素,区别在于如果要删除的元素不存在,remove()会报错,discard()不会。

clear():清空集合。

集合的应用很多,比如去除重复数据、比较两个文件的内容相似度、寻找文件或列表中只出现过一次的元素等,遇到跟集合相关的问题,灵活运用即可。



重点提示

在这一章中,你要掌握的内容:

(1) 熟练掌握字典的定义和操作。

(2) 明确键和值的意义。

(3) 理解字典的值可以为任何对象的意义。

(4) 掌握 Python 的集合的方法和应用。



动动手

(1) 利用字典实现四则运算小程序。

(2) 接收键盘输入的一串字符串,输出每个字符出现的个数,用字典编写计数器程序。

(3) 反向查找:给定一个字典的键,可以找到对应的值,但是,反过来根据值去找键却不行,而且可能有多个键映射到一个值上。现在,实现一个函数,根据给定的值返回对应的键,如果有多个,则返回一个列表。

进阶应用篇

第 8 章 文件和数据持久化

我们已经尝试编写了一些程序,但到目前为止这些程序都是瞬态的,因为这些程序在运行的过程中会产生一些数据,但随着程序结束,数据也会消失,再次运行,程序又从头开始了。

如果你已经写了自己的《英雄无敌》的程序,那你肯定已经意识到这个问题。比如玩家其实并没有真正的注册,即便这一次游戏运行过程中起了名字,玩的过程中也有了经验的增长或者各种变化,一旦中途退出,下次再启动游戏还要从头开始起名字,从起点开始游戏。

这时我们就需要数据的持久化,比如存储一部分关键数据到永久存储介质中(如硬盘),这样再次运行后,就可以从上次退出的地方继续运行,相当于游戏有了注册和保存进度的功能。

为了长期保存数据,方便修改和被其他程序使用,通过文件存取数据是一个比较常见的方法。当然,在计算机世界还有一个更强大的服务于数据的角色——数据库,比如 MIS(管理信息系统)就是使用数据库。但是一般应用程序的配置信息都是使用文件来存储的,我们这个单机版的《英雄无敌》现阶段更适合用文件存储。

除了简单的文本,也可以将程序的状态保存下来,下面介绍有关数据持久化的模块的内容。

8.1 文件读取



文件读写

文件分文本文件和二进制文件,它们都是由字节组成的信息,通常保存在存储介质上(如硬盘、U 盘)。文本文件都是可读的,比如可以通过文本编辑器直接打开或者在浏览器中显示。除了文本文件之外的都是二进制文件,比如图片、Word 文档,如果通过文本编辑器打开,会看到类似乱码一样的内容,不是可读字符。

Python 3 中提供了一个功能非常全面的 open()函数,用来打开文件、进行读写等操作。

需要说明的是,你要适应程序的运行状态,说是打开,其实并不像你在操作系统中打开文件就能直接看到内容。在 Python 中要访问一个文件,首先是跟文件建立一个链接,就像是在程序和磁盘上的信息之间建立了一个管道(通道),程序通过管道对磁盘上的数据进行读写操作。管道在 Python 中就是一个文件对象,open()函数用来创建文件链接的管道并返回代表这个管道的文件对象,通过这个对象的方法实现后续操作。文件对象还有一些其他的名字,比如句柄、文件流或文件描述符。

现在我们先准备一个文本文件，并且跟程序放在同一个文件夹中(这点很重要，在 8.5 节文件名和路径中有详细说明)。

```
#tmp.txt
First
Second
Third
```

文件的操作基本流程：打开→操作→关闭。

下面我们来打开文件并遍历：打开用 open()函数，open()的参数依次为：要打开的文件名，打开的方式'r'(读)；文件对象名为 f；通过 for 进行遍历操作，对文件对象进行遍历时，每次读取一行；最后用 close()方法关闭文件对象(管道)。

```
#fileTest_r.py
f = open("tmp.txt", 'r')
for line in f:
    print('=>',line)
f.close()
```

运行结果如下：

```
=> #tmp.txt
=> First
=> Second
=> Third
```

文件读取有 3 个方法：read()、readline()、readlines()。

1. read ( )

除了通过 for 遍历，文件对象自身还有很多读取文件的方法，比如，一次性读取文件全部内容的 read()，以单个字符串形式返回文件全部内容。

```
#fileTest_read.py
f = open("tmp.txt", 'r')
contents = f.read()
for line in contents:
    print('=>',line)
f.close()
```

因为是一个大字符串，所以遍历的时候会逐个字符的遍历，其结果如下(显示内容省略了 N 行)：

```
=> #
=> t
=> m
=> p
=> .
```

```
=> t
=> x
=> t
=>                    #注意这里
=> F
=> i
=> r
=> s
=> t
=>
```

这时你会发现每一行结束的位置多打了一个看不见的内容，这就是换行符，在不同的操作系统这个符号会有些区别，比如'\n'(UNIX)、'\r\n'(MS Windows)、'\r'(Mac)，可能会导致同样的代码跨不同平台使用时会产生不同的结果。Python 通过'U'修饰符来解决跨平台问题，比如 open('file. txt', 'rU').

2. readline ( )

如果需要逐行读取文本内容就可以通过 readline()方法，每调用一次读取一行。

```
#fileTest_readline.py
f = open("tmp.txt", 'r')
line1 = f.readline()
print(line1)
line2 = f.readline()
print(line2)
for line in f:
    print('=>',line)
f.close()
```

运行结果如下：

```
#tmp.txt
First
=> Second
=> Third
```

从结果可以看到，每次 readline()会读取一行，当执行两次后再去直接遍历文件对象会从 readline()结束的位置开始，这是因为文件操作过程中有个看不见的指针，后面会有说明。

3. readlines ( )

一次性读取多行，返回一个列表，每个元素是一行，这个的用法相信你现在应该可以看帮助尝试使用了。

8.2 文件写入

以'r'模式打开的文件相当于只读，是不能写入的，如果打开的文件不存在，还会报错。要想向文件写入数据则需要其他模式的参数，'w'和'a'都是可以写入的模式，并且若文件不存在不会报错，而是直接创建一个。具体模式及相应的作用如表 8-1 所示。

表 8-1　文件模式

模式	打开方式	文 件 存 在	文件不存在	打开后指针位置
'r'	只读	打开	报错	文件开始处
'w'	只写	清空文件内容	创建并打开	文件开始处
'a'	只写	打开	创建并打开	文件末尾
'r+'	读写	从开始处打开，或替换数据	报错	文件开始处
'w+'	读写	清空文件内容	创建并打开	文件开始处
'a+'	读写	打开	创建并打开	文件末尾

不同的打开方式会产生不同的效果，比如'r+'和'a+'都具备读写模式，但是指针位置不同就导致'r+'实现的是重写的效果，'a+'实现的是追加新内容的效果。还有'w+'，既然是清空文件内容重新开始写，文件是空的，“读”有什么意义呢？其实，在程序对文件的操作过程中写入数据后就可以读了，但是要注意因为指针的原因，写完就读是不会有数据的，需要把指针向前移动，下节将详细说明。

文件对象写入的方法也有 3 个：write()、writeline()、writelines()。

write()一次写入指定字符串，如果希望是多行数据，则需要在字符串中通过转义字符控制所有格式。

```
#fileTest_write.py
s = "hello\nworld"
f = open("wtmp.txt", 'w')
f.write(s)
f.close()
```

文件内容如下：

```
hello
world
```

writeline()写入的内容会作为一行，每执行一次增加一行。writelines()的参数是一个元素为字符串的集合，通常会用元组或列表，每个元素都要是字符串。具体效果请动手试一下吧。

8.3 文件内的指针

在对文件内容操作时，我们可以从任意位置开始读取和写入（如果写入位置后面有内容则会覆盖），这需要通过 Python 提供的控制文件读写起止位置的方法 seek()，这个方法控制一个看不见的文件指针在文件内移动，定位到要读写的位置，读取和写入包括重写都需要注意指针的位置。

fileObject.seek(offset[, whence])：offset 是开始的偏移量，代表需要移动偏移的字节数；whence 为可选项，默认值为 0，给 offset 参数一个定义，表示要从哪个位置开始偏移，0 代表从文件开头开始算起，1 代表从当前位置开始算起，2 代表从文件末尾算起。下面在交互模式下以 tmp.txt 为例看一下指针的变化。

```
>>>f = open('tmp.txt', 'r')
>>>f.tell()             #当前指针位置
0
>>>f.read()
'#tmp.txt\nFirst\nSecond\nThird'
>>>f.tell()
30
>>>f.seek(0)            #从文件头移动 0 位
0
>>>f.tell()
0
>>>f.seek(10,0)         #从文件头向后移动 10 位
10
>>>f.read()
'First\nSecond\nThird'
>>>f.seek(-10,2)        #从文件尾向前移动 10 位
Traceback (most recent call last):
  File "<pyshell#166>", line 1, in <module>
    f.seek(-10,2)
io.UnsupportedOperation: can't do nonzero end- relative seeks
```

演示中的 tell()方法用来获取指针当前位置，通过 read()和 seek()移动指针位置，最后的 f.seek(－10,2)失败是因为在文本文件中，没有使用'b'模式选项打开的文件，只允许从文件头开始计算相对位置，从文件尾计算时就会引发异常。这里只要加上'b'模式就可以了。

```
>>>f = open('tmp.txt','rb')
>>>f.seek(- 5,2)
25
>>>f.read()
b'Third'
```

8.4 文件关闭

文件对象调用 close()方法意味着关闭管道,取消程序和文件的链接。关闭后就不能够再进行读写操作了。

需要注意的是,建立管道后的所有读写操作都是在内存上进行的,比如文件在硬盘上,我们打开文件读取后,此时读取的数据都在内存中,反过来写入时,也都是在内存中,即当你执行了 write()后,此时打开硬盘上的相应文件会发现,并没有写入新的内容,只有关闭文件对象后才会将内存中的数据同步到磁盘上。所以,如果忘记关闭文件可能会造成写入数据丢失。

如果想在不关闭文件的情况下同步数据,可以通过文件对象的 flush()方法。

```
>>>f = open('tmp.txt', 'a')
>>>f.write("hello!")
>>>f.flush()
>>>f.close()
```

执行过 f.flush()就可以打开文件看内容的变化了。

文件处理需要获取一个文件句柄,从文件中读取数据,然后关闭文件句柄。如果不关闭可能会有麻烦。对于这种事先需要设置、事后需要做清理工作的场景,Python 的 with 语句提供了一种非常方便的处理方式。

如果不用 with 语句,代码如下:

```
f = open("tmp.txt", 'r')
for line in f:
    print('=>',line)
f.close()
```

上述代码运行没有问题,但相比 with 语句稍显冗长,with 语句除了有更优雅的语法,还可以很好地处理上下文环境产生的异常(with 语句所求值的对象必须有一个__enter__()方法和一个__exit__()方法,类似方法在第 9 章有详细讲解)。下面是 with 版本的代码,不需要手动 close:

```
with open("tmp.txt", 'r') as f:
    for line in f:
        print('=>',line)
```

8.5 文件名和路径

至此有可能到你会碰到一些问题,比如打开文件失败,或写完文件后找不到文件在哪里,像下面这样:

```
>>>f = open('abc.txt')
Traceback (most recent call last):
  File "<pyshell#172>", line 1, in <module>
    f = open('abc.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

最后一行的异常说明了原因，即文件不存在。如果你碰到这样的问题又不知所措，那么这一节就可以给你答疑了。

操作系统的重要组成是文件和目录（文件夹）。目前的主流操作系统都是把文件放在目录结构中，每个目录都有3个属性：

(1) 目录中文件的列表。

(2) 目录中目录的列表。

(3) 包含父目录（当前目的上一级目录）的链接，用“..”表示。

每个文件的位置的表示方式被称为路径(path)，路径分相对路径和绝对路径。

(1) 相对路径：以当前目录为起点，当前路径可以用“.”表示。

(2) 绝对路径：从文件系统的顶层目录开始，比如Linux的根(/)，Windows的C盘。

所有的程序包括交互模式都有当前目录，在前面的例子中，我们直接给出文件名，Python就会在当前目录去寻找这个文件，如果当前目录没有这个文件，就会报错。

Python的os模块提供了用于操作系统和目录的一些相关函数，比如os.getcwd()就能返回当前目录。

```
>>>import os
>>>os.getcwd()
'C:\\CrazyPythonZeroToHero\\code\\8'
>>>print(os.getcwd())
C:\CrazyPythonZeroToHero\code\8
```

这里获得的是当前目录的绝对路径，可以看出这是Windows的一个路径，之前的tmp.txt就放在C:\\CrazyPythonZeroToHero\\code\\8这个路径下，也就是名为8的文件夹里。路径中的“\”用来分隔上下级，为防止转义字符的特殊含义，os.getcwd()对返回的字符串做了转义处理，所以会看到两个“\”。路径分隔符在Linux和苹果操作系统中是“/”。如果要通过绝对路径打开tmp.txt就需要这样表示：

```
>>>open('C:\\ CrazyPythonZeroToHero\\code\\8\\tmp.txt')
```

直接提供文件名的形式就是相对路径的写法，Python会假设该文件位于当前程序所运行的目录中。如果是上一级或下一级目录，相对路径可以这样表示：

```
path = "./code/tmp.txt"     #当前目录下的 code 目录下的 tmp.txt 文件
path = "../tmp.txt"         #当前目录的父目录中的 tmp.txt 文件
```

上述例子中的“.”代表当前目录，“..”代表父目录。

什么时候用绝对路径，什么时候用相对路径呢？如果你是在当前目录运行程序或启动shell，就可以通过相对路径打开以当前目录做参照的任何文件；如果需要指定确切的文件

位置，就必须提供绝对路径。

8.6 os 模块

os 模块提供了非常丰富的函数，这里我们先认识其中的一小部分。

首先是第一个应用，获取当前目录下某文件的绝对路径：

```
>>>os.path.abspath('tmp.txt')
'C:\\CrazyPythonZeroToHero\\code\\8\\tmp.txt'
```

接下来，打开的文件如果不存在也会报错，所以可以先判断一下给定的文件或目录是否存在：

```
>>>os.path.exists('test.txt')
False
>>>os.path.exists('tmp.txt')
True
```

如果存在，就是下一个问题了。这一串字符串最后如果不是后缀名，根本不能判断是文件还是目录，所以下面需要判断一下是文件还是目录：

```
>>>os.path.isdir('tmp.txt')        #是否为目录
False
>>>os.path.isfile('tmp.txt')       #是否为文件
True
```

最后把这几个函数合理利用一下，实现一个遍历给定目录，打印所有文件的绝对路径，先创建供测试的目录，结构如图 8-1 所示。

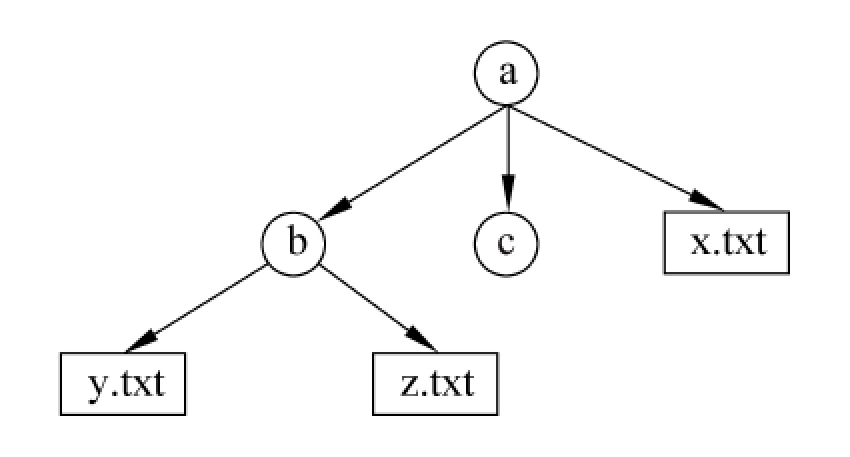


图 8-1 目录树

下面代码中 listdir()的作用是返回指定目录中文件和目录的名字组成的列表；接收 join()一个目录和一个文件名字，并将它们拼接成一个完整的路径。

```
#walkdir.py
import os
def walk(dirName):
    for name in os.listdir(dirName):
        path = os.path.join(dirName, name)
```

```
        if os.path.isfile(path):
            print(path)
        else:
            walk(path)

walk('a')
```

运行结果如下：

```
a\b\y.txt
a\b\z.txt
a\x.txt
```

这个例子中用到了函数的一种特殊用法，就是在 if…else 语句中，调用了函数自己，这种用法叫递归。

我们从逻辑上分析一下递归的意义，自定义函数 walk()的作用是获取目录下所有文件和目录的列表，遍历并获得完整路径，判断是否为文件，如果是文件，则直接打印，如果不是文件，那就是目录，然后呢？目录下可能还有目录和文件，所以现在要做的事就是再重复 walk()要做的事，所以直接调用自身。

递归函数一定要有结束的条件，否则就会进入到很深的递归中，本例的递归结束条件就是没有目录。

最后，其实 os 模块自带了一个 walk()函数，你可以通过帮助或查看手册搞明白使用方法，它要比这个自定义的函数强大方便得多，试着改写这个函数吧。

8.7 捕获异常

当我们尝试读取或写入文件时很有可能会出现错误，比如：

```
>>>open('abc.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

这时可以通过类似 os. path. exists 和 os. path. isfile 这样的函数，将可能出现错误的因素进行提前判断。但要考虑到所有因素会耗费大量的代码成本，这时我们可以尝试执行可能出现问题的代码，当发生问题时再解决，这个尝试就是 try 语句，语法类似 if：

```
try:
    f = open('abc.txt')
    for line in f:
        Pass
except:
    print("文件不存在")
```

Python 执行到 try 语句后，如果代码块正常执行，则跳过 except 语句并继续执行其他代码。如果发生异常，则跳出 try 并执行 except 语句下的代码块。

异常的类型我们在前面的练习中已经碰到了一些，可以在你的程序中尝试加入异常

处理。

8.8 数据序列化

很多时候我们会有下面的需求：

(1) 把内存中各种数据类型的数据通过网络传送给其他机器或客户端。

(2) 把内存中各种数据类型的数据保存到本地磁盘持久化。

我们可以把字符串写入文件中保存起来，但是要保存其他对象呢？比如列表、元组、字典等。Python 中提供了很多保存对象的模块，可以根据需要灵活选择。

将对象转换为可通过网络传输或可以存储到本地磁盘的数据格式(如 XML、JSON 或特定格式的字节串)的过程称为序列化；反之，则称为反序列化。

8.8.1 pickle 模块

pickle 模块实现了用于对 Python 对象结构进行序列化和反序列化的二进制协议，pickle 模块序列化和反序列化的过程分别叫 pickling 和 unpickling。

(1) pickling：将 Python 对象转换为字节流的过程。

(2) unpickling：将字节流二进制文件或字节对象转换回 Python 对象的过程。


序列化：pickle

pickle 模块可以将几乎所有类型的对象转换为适合保存的字符串写入文件中，并且可以转换回来直接变成对象。

通过 pickle 把数据持久化到本地磁盘，这部分数据通常只是供系统内部使用，因此数据转换协议以及转换后的数据格式也就不要求是标准、统一的，只要本系统内部能够正确识别即可。但是，系统内部的转换协议通常会随着编程语言版本的升级而发生变化(改进算法、提高效率)，因此通常会涉及转换协议与编程语言的版本兼容问题。

现在来看一下通过 pickle 模块进行序列化和反序列化的例子。

序列化常用方法有下面两个。

(1) pickle.dumps(object)：接收一个对象作为参数，并返回对象的字符串表达形式。

(2) pickle.dump(object,f)：用来将对象 object 写入二进制文件 f 中，第一个参数 object 是对象名，第二个参数 f 是文件对象。写入的数据同 pickle.dumps(object)返回的字符串。

```
>>>import pickle
>>>f = open('test1.pickle', 'wb')
>>>t = (1,2,3)
>>>pickle.dumps(t)     #返回元组 t 的二进制字符串形式
b'\x80\x03K\x01K\x02K\x03\x87q\x00.'
>>>pickle.dump(t, f)   #对象写入文件
>>>os.listdir()        #列出点前目录的文件列表，此时未同步到磁盘
['a', 'fileTest_r.py', 'fileTest_read.py', 'fileTest_readline.py', 'fileTest_
write.py', 'filetmp.txt', 'test1.pickle', 'tmp.txt', 'tmp_bak.txt', 'walkdir.py',
'wtmp.txt']
```

```
>>>f.flush()            #同步
>>>os.listdir()         #已写入磁盘
['a', 'fileTest_r.py', 'fileTest_read.py', 'fileTest_readline.py', 'fileTest_
write.py', 'filetmp.txt', 'test1.pickle', 'tmp.txt', 'tmp_bak.txt', 'walkdir.py',
'wtmp.txt']
>>>open('test1.pickle', 'rb').read()b'\x80\x03K\x01K\x02K\x03\x87q\x00.'
#直接读取没有意义,内容和 dumps 一样
```

现在已经成功地将对象写入磁盘的一个二进制文件中了,想要使用这个文件中保存的对象时,只需要通过 pickle.load(f)方法从二进制文件 f 中读取对象的字符串再还原成对象,也就是反序列化:

```
>>>import pickle
>>>f = open("test1.pickle", 'rb')
>>>newT = pickle.load(f)
>>>print(newT)
(1, 2, 3)
```

8.8.2　json 模块

如果要将一个系统内的数据通过网络传输给其他系统或客户端,通常需要把这些数据转化为字符串或字节串,而且需要规定一种统一的数据格式才能让数据接收端正确解析并理解这些数据的含义。XML 是早期被广泛使用的数据交换格式,在早期的系统集成论文中经常可以看到它的身影;如今大家使用更多的数据交换格式是 JSON(JavaScript Object Notation),它是一种轻量级的数据交换格式。JSON 相对于 XML 而言,更加简单、易于阅读和编写,同时也易于机器解析和生成。除此之外,我们也可以自定义内部使用的数据交换格式。


序列化:JSON

大部分编程语言都会提供处理 JSON 数据的接口,Python 2.6 开始加入了 json 模块,且将它作为一个内置模块提供,无须下载即可使用。

Python 的 json 模块序列化与反序列化的过程分别叫 encoding 和 decoding。

(1) encoding:把 Python 对象转换成 JSON 字符串。

(2) decoding:把 JSON 字符串转换成 Python 对象。

Python 与 JSON 数据类型转化关系如表 8-2 所示。

表 8-2　Python 与 JSON 数据类型

Python	JSON	Python	JSON
dict	Object	True	true
list, tuple	array	False	false
str	string	None	null
int, float, int- & float-derived Enums	numbers		

json 模块的常用函数跟 pickle 是一样的,可以直接参考 pickle 演示的例子,模块改成

import json 就可以了。

pickle 模块与 json 模块对比如下。

(1) JSON 是一种文本序列化格式(它输出的是 unicode 文件,大多数时候会被编码为 utf-8),而 pickle 是一个二进制序列化格式。

(2) JOSN 是我们可以读懂的数据格式,而 pickle 是二进制格式,我们无法读懂。

(3) JSON 是与特定的编程语言或系统无关的,且它在 Python 生态系统之外被广泛使用,而 pickle 使用的数据格式是特定于 Python 的。

(4) 默认情况下,JSON 只能表示 Python 内建数据类型,对于自定义数据类型需要一些额外的工作来完成;pickle 可以直接表示大量的 Python 数据类型,包括自定义数据类型。

(5) 需要与外部系统交互时用 json 模块。

(6) 需要将少量、简单 Python 数据持久化到本地磁盘文件时可以考虑用 pickle 模块。

如果需要将大量 Python 数据持久化到本地磁盘文件或需要一些简单的类似数据库的增、删、改、查功能时,可以考虑用 shelve 模块。

shelve 是一个简单的数据存储方案,类似 key-value 数据库,可以很方便地保存 python 对象,其内部通过 pickle 协议实现数据序列化。我们可以把 shelf 对象当 dict 使用,存储、更改、查询某个 key 对应的数据。

shelve 只有一个 open()函数,这个函数用于打开指定的文件(一个持久的字典),然后返回一个 shelf 对象。shelf 是一种持久的、类似字典的对象。操作很简单,可以通过手册自学。

8.9 CSV 文件

CSV 也叫逗号分隔值(分隔字符也可以不是逗号),其文件以纯文本形式存储表格数据,后缀名是 *.csv。但 CSV 并不是一种独立的文件类型,而是一种文件格式,CSV 实际上就是文本文件,可以通过 open()操作。另外,用记事本就可以直接编辑,当然用 Excel 也可以,因为 CSV 也是表格的变种。CSV 文件由任意数目的记录组成,记录间通常用逗号分隔;每条记录由若干字段组成。通常,所有记录都有完全相同的字段序列。

CSV 是一种通用的、相对简单的文件格式,被用户、商业和科学广泛应用。最广泛的应用是在程序之间转移表格数据。例如,一个用户可能需要交换信息,从一个以私有格式存储数据的数据库程序到一个数据格式完全不同的电子表格。最有可能的情况是,该数据库程序可以导出数据为 CSV,然后被导出的 CSV 文件可以被电子表格程序导入。


CSV

8.9.1 CSV 模块

首先生成一个 CSV 文件,这里有一个普通的 Excel 表格文件 testcsv.xlsx,内容如图 8-2 所示。

只需要另存为 csv 类型就可以了,用记事本打开就会开看到如图 8-3 所示的内容。

可以看到,CSV 文件就是普通文本文件,表格最后一行空白的位置也由逗号隔开,但是表格中的共识是没办法实现的,所以并不会改一个值而影响到其他值。看到这个文本,相信你已经可以对 CSV 文件进行操作了,Python 还提供了一个 CSV 模块,能够以相对简单

	A	B	C	D	E
1	name	mathematics	english	chinese	average
2	milo	100	100	100	100
3	tom	60	0	30	30
4	jerry	80	80	80	80
5	average				70

图 8-2　表格内容

test.csv - 记事本

文件(F)　编辑(E)　格式(O)　查看(V)　帮助(H)

```
name, mathematics, english, chinese, average
milo, 100, 100, 100, 100
tom, 60, 0, 30, 30
jerry, 80, 80, 80, 80
average, , , , 70
```

图 8-3　CSV 文件内容

的方式读写 CSV 文件。

8.9.2　CSV 读写

csv 模块提供了 csv. reader(fileobject)读取文件，以及 csv. writer(fileobject)写 CSV 文件。参数是文件对象，这意味着要先打开文件后再进行 CSV 读写操作。

csv. reader 会遍历文件，每次循环返回文件中的一行，并以字符串列表的形式返回，列表中每个元素就是表格列的值，例如：

```
#csvtest.py
import csv
fobj = open('test.csv','rU')  #注意 U
csvobj = csv.reader(fobj)
for row in csvobj:
    print(row)
```

运行结果如下：

```
['name', 'mathematics', 'english', 'chinese', 'average']
['milo', '100', '100', '100', '100']
['tom', '60', '0', '30', '30']
['jerry', '80', '80', '80', '80']
['average', '', '', '', '70']
```

获取数据后我们来更新一下表格数据。写入数据时，并不能像在表格中那样，选一个，然后直接更改，而是要把原数据全部读取出来，更改指定值后再写回文件中，比如，将 milo 的所有期末考试成绩变成 70 分，平均分就变成了 90 分，操作如下：

```
#csvtest.py
```

```
import csv
readObj = open('test.csv','rU')
csvobj = csv.reader(readObj)
allCsv = []
for row in csvobj:
    allCsv.append(row)

allCsv[1][3] = 70

theSum = 0
for i in allCsv[1][1:4]:
    theSum += int(i)
x = theSum/3
allCsv[1][-1] = '%d' %x

theSum = 0
for row in allCsv[1:-1]:
    theSum += int(row[-1])
y = theSum/3
allCsv[-1][-1] = y

with open('newtest.csv','w') as writeObj:
    writer = csv.writer(writeObj)
    for row in allCsv:
        writer.writerow(row)
```

需要注意的是，程序运行过程中，不要同其他应用程序打开或占用 CSV 文件，写入的时候要写入一个新的文件。结果如图 8-4 所示。



图 8-4　结果

最后的平均数比较长，请你改写程序使小数点后保留两位。

重点提示

在这一章中，你要掌握的内容：

(1) 掌握文件的读写操作。

(2) 明白文件内指针的作用。

(3) 理解路径的意义。

(4) 会进行数据持久化操作。



动动手

(1) 编写程序,可用来复制文件,比如文本文件、图片、音频、视频。提示:二进制形式从源文件读取,写入目标文件。

(2) 自建文本文件,内容有 hi,替换文件中 hi 为 hello。提示:每次读两个字符。

(3) 编写程序,由用户输入文件名,在打开文件时,如果不存在,实现反复提示,要求输入正确的文件名,要求用到异常处理。

(4) 利用 CSV 实现一个学生信息管理系统,初始状态为每个学生有三条记录:学号、姓名、电话。程序须提供的功能包括:显示数据、删除行、插入行、修改单元格数据。

提示:① 读取表格数据后要以适当的数据结构存储;② 各操作以字母作为开启接口,比如 I 代表要插入数据。

(5) 考虑给《英雄无敌》增加相应功能:

① 存进度(保存玩家信息、角色状态等)。

② 程序启动后,用户可以通过用户名和密码登录游戏。

③ 对用户名和密码进行验证,3 次机会,验证失败,则锁定账号,由管理员解锁。提示:锁定功能可以用创建一个文件的形式,比如用户被锁定后则创建一个文件,下次用户登录时,程序检测到这个文件,则代表用户被锁定,管理员手动删除此文件即可解锁,也可以写一个管理员解锁专用程序。



《英雄无敌》:软件上锁

具体的实施可以根据自己的想法来实现,遇到问题解决问题,实现不了的就换思路。

第 9 章 面向对象

我们已经系统学习了用 Python 中的内置类型和基本语法组织数据与程序。比如可以通过列表把数据收集起来,可以通过函数把代码封装起来以便反复使用。

越深入学习,我们就越希望可以提高代码的重用度,比如函数、模块。但是,当我们需要编写一个较为庞大的项目时,单纯的数据或者函数已经满足不了实际需求,这时就需要引入面向对象这个新的编程模式,或者称之为一种思想。

在前面的学习中已经接触过对象了,最简单的比如一个字符串对象,它其实就是把数据和函数都收集到了一起。对象的函数就是方法,用 help(str)就能看到所有字符串对象的方法,通过点记法就可以调用,比如"hello". upper()。

在这一章我们要讲的是什么是对象,如何创建和使用对象。

9.1 从《英雄无敌》开始认识对象

对于新手,面向对象可能不太好理解,但是也没有那么难,就跟学习如何定义字符串没什么区别,不用太着急开始,可以先重新思考一下设计的《英雄无敌》游戏,不知道你是否已经设计了一个很不错的文本模式的游戏,现在可能要全部重新设计了。



《英雄无敌》抽象化开发

现在按照我的思路来设计这个游戏:首先至少需要 2 个角色,一个是英雄,一个是怪兽,每个角色可能有各种不同的样子,比如每个玩家的英雄人物不同,且英雄和怪兽都有不同的技能,如人拿刀砍怪兽,怪兽可以咬人,怎么描述这两种不同的角色和他们的共同功能呢?

按我们已经掌握的知识,写出了如下的代码来描述这两个角色:

```
#定义英雄和怪物的模板
def hero(name, hp, attack):
    data = {
        'name':name,
        'hp':hp,
        'attack':attack
    }
    return data

def monster(name, hp, monster_type):
```

```
    data = {
        'name':name,
        'hp':hp,
        'type':monster_type
    }
    return data
```

上面两个函数相当于造了两个模板,游戏里的每个英雄和每个怪物都拥有相同的属性(变量)。游戏开始后,根据每个玩家或某个怪物传入的具体信息来塑造一个具体的英雄人物或者怪物,方法如下:

```
hero2 = hero('mario', 100, 20)
monster1 = monster('turtle', 200, 'turtle')
```

这样就相当于有了两个角色对应的对象,接下来就是他们各自的方法,英雄可以攻击怪兽,怪兽可以溜达着咬英雄,只须分别定义对应的函数,需要谁做什么调用谁的函数就可以了,代码如下:

```
def kill(h):
    print("hero %s attacking monster!" %h['name'])
def walking(m):
    print("monster %s : Walking!"%m['name'])

kill(hero2)
walking(monster1)
```

运行效果如下:

```
hero mario attacking monster!
monster turtle : Walking!
```

看起来已经实现了我们要的效果,有英雄和怪物对象,各自有各自的方法,但实际上,对于这两个方法你可以看到,它们并不是谁专属的,因为下面这样也是可以的:

```
>>>kill(monster1)
hero turtle attacking monster!
```

可以看到,代码运行正确,但是这看起来就像是把怪兽对象传给了英雄的方法,这在游戏逻辑上是完全不允许的。这时就需要人为在代码上加以限制,代码如下:

```
#hero2.py
#定义英雄和怪物的类别函数
def hero(name, hp, attack):
    def kill(h):
        print("hero %s attacking monster!" %h['name'])
```

```
        data = {
            'name':name,
            'hp':hp,
            'attack':attack,
            'kill':kill
        }
        return data

    def monster(name, hp, monster_type):
        def walking(m):
            print("monster %s : Walking!"%m['name'])

        data = {
            'name':name,
            'hp':hp,
            'type':monster_type,
            'walking':walking
        }
        return data
```

这样，从代码结构上我们已经把各自的方法区分开了。其实，类似这样的做法就已经近似面向对象的思想了。最早开始出现面向对象编程的想法也是基于类似的问题发展起来的。

当然，这里只考虑了一点点，随着程序复杂度的增加，需要解决的问题会更多。不过，像这样将函数和变量封装到一起的结构就是类，有专门的关键字，在后面将详细介绍。现在你有一个初步的印象就好，不用着急弄懂所有的东西。

9.2 从面向过程到面向对象

之前所介绍的解决问题的方法都是面向过程的，所谓的面向过程就是解决问题的步骤，每一步要做什么都要安排好。

面向过程的好处是降低了写程序的复杂度，只要顺着程序员的思路写就可以一步一步解决问题；但缺点也很明显，就是在这个过程中只要有一步需要改动，有可能整个代码就得“大换血”。

习惯了面向过程，初学编程的人刚开始学习面向对象会有点别扭，主要是在理解上。面向对象的程序设计核心是对象，通过对象与对象之间的联系来完成任务。程序中什么都可以作为对象，需要什么创建什么，比如你要设计一个闯关游戏，先设计了一堆英雄人物，每个人都有各自的特征和技能，其实每个人都是对象，包含各自的属性（特征）和方法（技能）。只有英雄的游戏没有意思，就再设计一群怪物来捣乱，或者英雄之间也可争斗。总之，每个都是对象。程序员只需要创建这些对象，至于他们是怎么进行游戏并取得最终胜利的根本不用管。

面向对象的优点是降低了程序本身的复杂性，复杂性降低意味着 bug 少，bug 少就意味着可维护性高。缺点是可控性差，因为面向对象不像面向过程那样精准地预测问题处理的

流程与结果，面向对象是依靠对象与对象之间的关联来解决问题，所以，如果有个别对象失误，会导致程序运行出问题。

面向过程的开发更适合简单的或者稳定性高不需要太灵活的程序，而面向对象则适合庞大的程序的处理，后面我们介绍的很多工具都是面向对象开发的。

学习面向对象的程序设计，有些概念需要在头脑里先形成一个基本的印象，这样有助于面向对象的学习。

面向对象（简称 OOP）是现在比较普遍的编程方法，主要用于日益复杂的编程项目。OOP 其实不是什么语法结构，它是一个概念，即程序由对象组成，每个对象都可与程序中其他对象进行交互。同一类对象的数据和函数抽象打包到类中。

（1）类（class）：用来描述具有相同属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。类是对象的抽象化。

（2）类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。

（3）实例化（instantiation）：创建一个类的实例的动作，类的具体对象。

（4）实例变量：定义在方法中的变量，只作用于当前实例的类。

（5）对象/实例（object/instance）：对象是通过类定义的数据结构的实例。对象包括两个数据成员（类变量和实例变量）和方法。

（6）面向对象三大特性：封装、继承和多态。这三个特性会在后续介绍中详细说明，深入了解后能更准确地理解这三个特性的意义。

不用急于弄明白这些概念的意义，先在脑子里留下印象，在接下来的内容中我们会逐一学习。

9.3 类和对象

Python 遵循标准的面向对象编程模型，理解面向对象编程的途径很多，这是一个适应过程，不用急于理解所有概念。


类和对象

在这里我们对类的定义是建立新对象的模板，而利用这个模板所创建的对象称为这个类的实例，类和对象的关系好像汽车工厂和所制造的汽车，通常有一整套标准流水线，这个流水线可以生产出无数辆车，而且都具备相同的一些属性，比如颜色、材质、组成的零件等，也具备相同的方法，比如启动、前进、后退、刹车等。

类的概念可以联想到数据类型，我们讲过的大部分数据类型都可以看作模板，比如 str 是类型，'a'、'b'、'c'都是字符串类型的实例。在 str 这个类型中定义了字符串的属性和方法，所有字符串对象都可以调用这个类型当中的方法，不同的对象又都具备不同特点，比如各自的值。

现实中也可以找到各种各样的类以及与其对应的对象，你可以看看周围试着把它们抽象成类，比如看见电视、空调就可以抽象为家电类，属性就是颜色、材质等，方法就是开机、运行等。

现在我们来一步步创建一个类并且逐一认识其中的组成元素，Python 使用 class 关键

字创建类，基本的格式为

```
class 类名:
    语句
```

class 关键字后跟类名，类名命名规则同变量，如果有多个单词，习惯上首字母大写，冒号后代表进入类内部的语句，需要缩进。

先来看一个简单的类和实例化后的对象的工作过程，以便对类有一个整体印象。定义一个三角形类，这个类型的对象都具备三条边，对象都可以求自己的周长。

```
#triangle_class.py
class Triangle:
    def __init__(self, x, y, z):
        self.a = x
        self.b = y
        self.c = z
    def perimeter(self):
        return self.a + self.b + self.c

#main
t1 = Triangle(3,4,5)
t2 = Triangle(9,9,9)
print(t1.perimeter())
print(t2.perimeter())
```

结果如下：

```
直角三角形周长 = 12
等边三角形周长 = 27
```

在这个例子中，有

(1) class 定义了一个 Triangle 类。

(2) __init__叫作构造函数，用来初始化对象。

(3) __init__中定义了三个属性 a、b、c，前面必须加 self. 前缀，在类内部调用时必须使用 self 访问，在实例化对象时通过 x、y、z 传给对象。

(4) 类当中定义的函数 perimeter 就是对象的方法，形参中第一个必须是 self，用于在类内部传值。

(5) 主程序中通过类实例化了两个对象 t1 和 t2，通过调用 perimeter 方法得到对应的返回值。

9.4 属性和方法

属性和方法是类的两个成员，只面向对象编程特有的概念。属性就是类所封装的数据，方法即类对数据进行的操作。

9.4.1 类的属性



属性和方法

属性其实就是变量，跟变量一样有作用域，而且根据定义的方式不同会有不同的效果。通过一个例子认识一下类的属性。

```
>>>class Ren:                 #定义"Ren"类
        name = "Ren"          #类属性、公有属性
        __money = 0           #私有属性，前加双下划线


>>>print(Ren.name)            #通过类名可以直接调取类属性
Ren
>>>print(Ren.__money)         #调取私有属性失败
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print(Ren.__money)
AttributeError: type object 'Ren' has no attribute '__money'
>>>milo = Ren()               #实例化对象 milo
>>>print(milo.name)           #调取对象的公有属性
Ren
>>>milo.name = "milo"         #对象属性赋值
>>>print(milo.name)
Milo
>>>print(milo.__money)        #调取私有属性失败
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    print(milo.__money)
AttributeError: 'Ren' object has no attribute '__money'
```

通过这个流程可以看到类属性、公有属性和私有属性。类属性就是公有属性可以被类直接调用，可以在类外被对象直接调用。私有属性在类外部无法直接调用（会报错），如果要使用私有属性，只能在类内部通过方法调用，私有属性实现是通过名字来区分的，名字以两个下划线开始。

私有属性的意义在于保护数据，当不希望在类外部对其属性进行操作时，就需要使用私有属性了。

除了公有属性和私有属性之外还有一种内置属性，是名字前后都有双下划线，通过 dir 能看到所有的属性：

```
>>>dir(Ren)
['_Ren__money', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'name']
```

通过这个结果可以看到，私有属性前面直接带了类的前缀，另外，像__doc__、__init__

这样的就是内置属性和方法。内置属性不需要定义，是在 Python 基本架构中就存在的，在下一节我们会介绍。

9.4.2 类的方法

属性是变量，方法其实就是函数，方法也分公有、私有和内置，只是在类中定义方法时第一个参数必须是 self。

首先来看内置方法，通过三角形类的演示，我们已经知道从类实例化出对象的方式，其实__init__并不是必须要编写的，__init__叫构造函数，是用来订制对象的初始状态。我们通过__init__来看一下 self 的重要性。

再次强调一下：定义一个类时，实际上是在定义一个订制工厂函数，然后可以在你的代码中使用这个工厂函数创建实例：

```
m = MyClass()
```

类名后的小括号就是告诉 Python 要创建一个对象，Python 在处理这行代码时，会把工厂函数调用转换为以下调用：

```
MyClass().__init__(m)
```

而我们在定义__init__时的写法是这样：

```
def __init__(self):
    #要初始化给对象的代码
```

通过这几步，你有注意到什么吗？ self 实际上就是对象本身，就是将 m 赋值给 self，这个赋值很重要，在之前用函数模拟面向对象思想时，没有这种参数，所以，实际上无法真正区分开对象。类的所有对象之间，属性不共享，方法共享，而类中的这个 self 就可以知道方法调用的对象是谁。

下面重新编辑一下 9.4.1 小节中的 Ren 类，将类属性和对象属性分开，并且定义相应的一些方法。

```
#class_ren.py
class Ren:
    """这里是关于类的描述"""
    className = "Ren"
    def __init__(self, name = '', money = 0):     #初始化对象属性
        self.name = name
        self.__money = money

    def say(self):                                 #公有方法
        """有必要的时候，这里是方法的说明"""
        print("I am %s ,i have %d yuan" % (self.name, self.__money))

#main
```

```
if __name__ == "__main__":
    milo = Ren()                          #直接实例化对象
    zqx = Ren("zouqixian", 100)           #传参实例化
    milo.say()
    zqx.say()
    print(Ren.className)
    print(Ren.name)                       #对象属性,类无法调用
```

运行效果如下：

```
Ren
I am  ,i have 0 yuan
I am zouqixian ,i have 100 yuan
Traceback (most recent call last):
  File
"C:/Users/milo/Desktop/CrazyPythonZeroToHero/code/9/class_ren.py", line 18, in <module>
    print(Ren.name)
AttributeError: type object 'Ren' has no attribute 'name'
```

通过这段代码可以看到以下几点：

（1）__init__内定义属性的代码，只有在实例化时才会执行，所以通过类直接调用时是不会成功的。

（2）私有属性可以在类内部通过方法调用，实例化对象后可以通过这样的方法对私有属性操作，取值或赋值。

（3）AttributeError 是类当中最常见的异常，属性和方法都会报这个异常。

私有方法的定义方式也是通过在名字前面加双下划线实现的，你可以实践一下。私有方法也不能在类外调用，只能在类内部调用。

9.4.3 内置属性和方法

Python 中提供了一些以双下划线开始且以双下划线结束的属性和方法，类中也有一些专属的属性和方法。比如__doc__，在上一个例子中写了一些类和方法的注释，可以通过__doc__来看到。

```
>>>print(zqx.__doc__)
#这里是关于类的描述
>>>print(zqx.say.__doc__)
#有必要的时候,这里是方法的说明
```

如果你直接打印一个对象，得到的结果会比较晦涩，如果想得到一个自定义的信息，可以利用__str__方法：

```
>>>class X:
    pass

>>>x = X()
```

```
>>>print(x)
<__main__.X object at 0x05D39650>
```

此时返回的是这个对象在内存上的地址，重写一下__str__方法：

```
>>>class X:
    def __str__(self):
        return """this is a instance of class X"""

>>>x = X()
>>>print(x)
this is a instance of class X
```

像__str__这样的方法还有个名字，叫魔术方法，因为自动化程度比较高，能实现一些看起来有点神奇的作用。

还有很多内置方法，就不一一介绍了，需要的时候拿来用就可以了。最后再说一个析构函数__del__。__init__是在实例化对象时自动执行，而__del__则是在释放对象时自动执行。通常用来做一些资源关闭的操作，比如文件、数据库。

9.5 类的继承

继承是面向对象的三大特性之一，简单地说就是一个新类可以通过继承来获得已有类的方法和属性，这个新类也可以自己定义新的方法和属性。



多态和继承

新类通常被称为子类，被继承类则称为父类，继承的格式如下：

```
class Son(Father[,Father...]):
    pass
```

继承的方式比较灵活，可以继承一个父类也可以继承多个。

9.5.1 使用继承

子类继承父类时，只能继承父类的公有属性和公有方法，不能继承父类的私有属性和私有方法。下面通过实例来看一下继承的效果。

```
#class_fs.py
class A:
    name = "class A"

    def __init__(self):
        self.i = "init A"

    def a(self):
        print("function a")
```

```
class B:
    name = "class B"

    def __init__(self):
        self.i = "init B"

    def b(self):
        print("function b")

class C(A):
    pass

class D(B,A):
    pass

#main
c = C()
print("c.name",c.name)
print("c.i",c.i)
c.a()

d = D()
print("d.name",d.name)
print("d.i",d.i)
d.a()
d.b()
```

运行结果如下：

```
c.name class A
c.i init A
function a
d.name class B
d.i init B
function a
function b
```

这个例子中我们分别定义了 A 和 B 两个类，C 继承了 A，D 继承了 B 和 A(注意这个顺序)。C 只重新定义了一个 name 属性，D 自身并没有定义新的属性，所以我们在后面看到的都是继承来的。

C 只继承了 A，所以也就继承了 A 的 name 属性以及方法，主程序中我们直接通过名字就可以调用相关属性和方法，就好像自己定义的一样，通过输出可以看出所有的值也都来自于 A，包括构造函数初始化的值。但是因为我们在 C 中重新定义了 name，所以 name 的值就是新值。

D 类继承了两个类，当继承多个类的时候，需要注意顺序问题，如果多个父类中有相同的属性或者方法，只继承第一个，不会被后面的覆盖。所以 A 和 B 类同样都有 name 属性，

我们看到的就是第一个继承的 B 的值。

9.5.2 重载

重载就是在子类中重新定义父类方法,因为很多时候从父类继承过来的方法并不能满足当前类的需要。不仅方法可以重载,运算符也可以重载,比如“+”“-”“*”“/”等,以适应子类进行相关操作。



方法重载

1. 方法重载

通过一个例子看一下方法重载。

```
#class_hb.py
class Human:
    name = ''
    __money = 0

    def __init__(self, name, money):
        self.name = name
        self.__money = money

    def show(self):
        print(self.name)
        print(self.__money)

class Baby(Human):                              #继承 human 类
    __height = 0                                #子类新定义的属性
    __weight = 0
    __sex = ""

    def __init__(self, name, money, height, weight, sex):     #重载__init__方法
        self.__height = height                  #子类对象的新属性
        self.__weight = weight
        self.__sex = sex
        Human.__init__(self, name, money)       #子类中调用父类方法

    def show(self):
        Human.show(self)                        #调用父类方法
        print(self.__height)
        print(self.__weight)
        print(self.__sex)

#main
x = Baby("milo", 100, 8, 60, "male")
x.show()
```

运行结果如下:

```
milo
```



```
100
8
60
male
```

在这个例子中，我们在Baby这个子类中定义了新的属性，重载了父类的两个方法，使其功能能够满足子类的需求。值得注意的是，在重载时相当于对父类继承过来的方法进行重新定义，就像变量重新赋值一样。所以，当我们重载父类方法后又需要使用父类方法时，可以通过"父类名.方法名"的方式直接调用。比如这个例子的两个重载方法中，我们都直接调用了父类的方法来获取父类定义的属性。

2. 运算符重载

在Python中，运算符也是通过相应函数实现的，换句话说，运算符对应的其实就是类中的一些魔术方法(专有方法)。比如，加、减、乘、除对应的就是__add__、__sub__、__mul__、__div__。可以在自定义类中重写这些方法来实现一些特殊的运算。

例如，想要实现一个自定义类，用来生成一个列表，然后重写加、减、乘、除的运算符，从而实现该类的对象的所有元素可以分别进行运算，以加法为例。

```
#class_add.py
class MyList:

    def __init__(self, *args):
        self.__myList = []
        for arg in args:
            self.__myList.append(arg)

    def __add__(self, x):
        """当对象通过 + 符号对后面数字运算就会调用此方法"""
        for i in range(len(self.__myList)):
            self.__myList[i] = self.__myList[i] + x

    def show(self):
        print(self.__myList)

#main
l = MyList(1,2,3,4,5,6,7,8,9,10)
l.show()
l + 10
l.show()
```

运行结果如下：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

当对象后面跟了"+"符号时，就会自动调用__add__方法。这样，我们重载__add__方

法就可以实现这个例子的要求了。

9.6 多态

面向对象的第三个特性就是多态，不过，多态并不是一个新的语法结构。简单地说多态指的是一类事物有多种形态，比如，一个抽象类有多个子类，因而多态的概念依赖于继承。

比如，序列类型就有多种形态：字符串、列表、元组。

假如定义一个形状类，形状类下有三角形类和正方形类等，在形状中有求周长的方法，子类中也必须要实现这个方法，而且子类中会有变化。即同一个名字的方法在不同类中有不同的作用，这种状态称为多态。

```
#shape_class.py
class Square:
    def perimeter(self):
        raise AttributerError("子类须重载此方法，否则抛出这个异常")

class Triangle(Square):
    def __init__(self, x, y, z):
        self.a = x
        self.b = y
        self.c = z
    def perimeter(self):
        print(self.a + self.b + self.c)

class Shape(Square):
    def __init__(self, x):
        self.x = x
    def perimeter(self):
        print(self.x * 4)

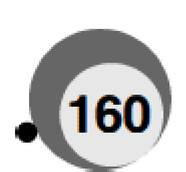
#main
triangle1 = Triangle(3,4,5)
shape1 = Shape(9)

triangle1.perimeter()
shape1.perimeter()
```

9.7 内置装饰器

装饰器比较神奇，作用的原理是在 Python 中可以将函数作为参数传递，这样就可以通过一个函数改变另一个函数的功能，这个能够改变其他函数的函数称为装饰器。

这里介绍三个类内置装饰器，实际上就是三个函数，用法上很简洁，也不需要知道这三个函数内部实现的方法。



（1）staticmethod（类静态方法）：与成员方法的区别是没有 self 参数，并且可以在类不进行实例化的情况下调用。

（2）classmethod（类方法）：与成员方法的区别在于所接收的第一个参数不是 self（类实例的指针），而是 cls（当前类的具体类型）。

（3）property（属性方法）：将一个类方法转变成一个类属性，即只读属性。

在介绍属性时说过类属性可以直接被类调用，如果要实现能被类直接调用的方法就可以借助 staticmethod 和 classmethod 了，区别在于 staticmethod 的方法没有 self 参数，通常用来直接定义一个静态类方法，如果想将一个普通动态方法变成类方法就要使用 classmethod 了。

```
#class_sc.py
class A:
    @staticmethod
    def sm():
        print("静态方法")

    @classmethod
    def cm(self):
        print("类方法")

a = A()
A.sm()
a.sm()
A.cm()
a.cm()
```

这里的用法有个@，比如@classmethod 要加在 cm 方法上面，这样的写法实际上是取代了 cm ＝ classmethod（cm），代码看起来更简洁明了。

property 的作用是将一个类方法转变成一个类属性，即只读属性。下面通过一个例子看一下实际运用。

```
#class_pp.py
class Triangle:
    def __init__(self, x, y, z):
        self.a = x
        self.b = y
        self.c = z
    @property
    def perimeter(self):
        return self.a + self.b + self.c

t = Triangle(6,6,6)
print(t.perimeter)
```

这样，只要经过 property 装饰器处理过之后，就可以将方法变成属性了，这在我们要处理方法的值时很方便。

9.8 《英雄无敌》面向对象设计


《英雄无敌》面向对象开发

面向过程的《英雄无敌》你写了吗？现在是颠覆的时候了。通过下面的例子，来看一下将类作为模块使用，这样可以将抽象代码和逻辑代码分开，我们编写一个模块定义一个英雄类和一个敌人类，然后看一下类和类之间的关联。一共两个文件，一个模块 heros. py，一个逻辑主程序 play. py。

现在准备开始了，如果你还没想好怎么开发模块，可以先想逻辑代码，比如，现在先写 play. py。

```
#game/play.py
from heroes import *                    #导入 heros 模块(1)
msg = "欢迎来到 英雄无敌的世界......!"
if __name__ == "__main__":
    print(msg)
    milo = Hero('milo')                 #实例化英雄(2)
    boss = Element('boss')              #实例化敌人(3)
    print("boss hp:",boss.hp)           #显示敌人的 HP(4)
    print("英雄发起攻击!")
    milo.hit(boss)                      #英雄调用 hit 方法攻击 boss 对象(5)
    print("boss hp:",boss.hp)           #显示敌人的 HP(4)
```

其实这个代码可以一点一点地写，每写一个功能就去模块中实现相应功能。比如第(1)步要导入 heros 模块，所以可以先创建这个模块文件，第(2)、(3)两步是要实例化对象，我们可以去模块中创建两个类。

```
#game/heros.py
class Hero:
pass

class Element:
    pass
```

通过步骤(2)、(3)、(4)、(5)，我们知道 Element 类至少需要一个 name 属性(初始化对象赋值)、一个 hp 属性，Hero 类需要一个 hit 方法，并且调用 hit 方法后会对 Element 的对象 hp 属性有影响。

```
#game/heros.py
class Hero:
    def __init__(self,name = 'player1',hp = 100,atk = 10):
        self.name = name
        self.hp = hp
        self.atk = atk
        print('英雄 %s 诞生!!'%self.name)
```

```
    def hit(self,name):
        name.hp -= self.atk

    def blood(self):
        pass

class Element:
    def __init__(self,name = 'boss',hp = 1000):
        self.name = name
        self.hp = hp
        print('BOSS %s 诞生!!'%self.name)

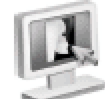
    def hit(self):
        pass
```

根据需要，我们创建了两个类，设置了相应的属性，增加了一个 atk 属性表示攻击力。需要注意一下 hit 方法，hit 方法的第二个参数实际上就是另一个对象，所以 name. hp 实际调用的是另一个对象的属性。

如果你细心观察会发现这两个类十分类似，最终 Element 也是要有一个 hit 方法的，所以这个模块还可以再抽象出一个父类，然后这两个类去继承，请你考虑一下如何实现。

至于这个小程序，模块写完不用动，只需要去执行 player. py 就可以了，效果如下：

```
欢迎来到 英雄无敌的世界......!
英雄 milo 诞生!!
BOSS boss 诞生!!
boss hp: 1000
英雄发起攻击!
boss hp: 990
```



重点提示

在这一章中，你要掌握的内容：

(1) 深刻理解面向对象程序设计。

(2) 能够熟练创建类和生成对象并调用属性和方法。

(3) 理解面向对象的三大特性：封装、继承、多态。

动动手

(1) 设计一个机动车类，需要包含属性和方法，再设计一个跑车类继承机动车类。

(2) 设计一个购物车类实现购物网站的购物车功能，假设是一个买书的网站，考虑如果有人买书要把什么东西放在购物车中，在购物车中可以执行哪些操作。

(3) 设计一个类，并且被作为模块用于对第 8 章的学生信息管理系统实现相应增、删、改、查操作。

(4) 面向对象同样有很多需要学习的东西，扫描





super 与新式类和经典类



内置装饰器

上页的二维码观看这两个视频，关于 super 与新式类和经典类，还有内置装饰器，了解一下吧。

(5) 软件编程中有一种模式叫测试驱动开发，意思就是先测试，后开发，如果你感兴趣，这里准备了一个视频，扫描右侧二维码即可观看。


测试驱动开发

(6) 利用面向对象思想重新设计游戏《英雄无敌》，可自行考虑游戏效果，以下为参考。完善《英雄无敌》(或实现类似功能的)游戏，游戏流程及功能如下。

① 开始游戏时，要求：给游戏添加一个登录、注册的功能，可以多账号，不同账号不同进度和等级。注册账号名字为邮箱地址，需要验证。密码为 6～16 位字母和数字的组合，不能全为数字或字母。需要将账号密码保存到一个文件当中，最好再做一个存取游戏进度的功能。

② 进入游戏后，由用户决定角色名字；主角有 100HP 血，10 点攻击力。

③ 游戏开始后有一个顺序十格地图，每一格分左右两个房间。

a. 除起点外在游戏中主角在每一个房间行走过程中会出现随机事件。比如加血，踩地雷掉血，增加攻击力(每次 5 点)，遇到 20HP 血的小怪兽(不论输赢，进行三回合战斗，互相有损伤，跟攻击力挂钩)。

b. 走到第十步时是大 Boss(2000HP 血)。

c. 双结局任务设置，比如用户进左边的房间多，则回合式战斗；如进右边房间多，则时间式战斗。

回合式战斗指每人一次攻击，HP 血先为 0 的一方输。

时间式战斗指有 30 秒时间，每 2 秒掉 20HP 血，30 秒内 Boss HP 血为 0，则主角胜利，游戏过程中主角 HP 血降到 0 时，game over。

第 10 章

异常处理

异常就是程序运行过程中引发的错误。相信你在前面的练习和实验中已经遇到过很多次,在前面的内容中我也有意提示你留意出现过的异常,如果你这样做了,应该会发现异常并不是每个都不同,经常出现的就是有限的那几个。


异常处理

比如,在序列操作中会碰到索引越界,打开的文件不存在,做除法除数为 0 等,通常遇到这些问题时,程序会自动终止,我们在之前通过条件语句可以适当避免一些,但是很有限。在部分例子中也介绍了异常处理的方法,即 try,这也是程序员用来解决问题的最好方法。

Python 中提供了非常完善的异常处理机制。异常处理可以帮助程序员提高程序的健壮性,并不是预防异常不让它发生,而是在异常发生时,系统不会强行终止程序;是执行预设的异常处理代码,执行完毕后,程序还会继续执行下去。所以掌握异常的处理对于程序员来说是很重要的。

10.1 异常

学习异常处理,首先是要正确认识程序出现的错误,有些是不属于异常处理范围的。程序运行常见错误主要有 3 类:语法错误、逻辑错误及运行错误。

(1) 语法错误就是我们在讲基础语法时所说的那些不符合语法规则的语句所产生的错误,比如数据类型错误、符号错误、关键字不匹配、表达式不完整等,Python 在运行程序时会对这些语法错误进行检测并诊断,其实这些错误都是需要程序员进行检查纠正的,这些都不属于异常处理范围。

(2) 逻辑错误是指程序运行的过程和结果没有按照程序员所设想的方式执行,比如条件语句或者循环语句设计错误,代码执行的先后顺序不对,设计的算法不对等,这些也不属于异常。

(3) 运行错误是程序运行过程中除以上两个问题之外出现的错误,比如文件打不开、对象未定义、索引越界等,这些都是异常。

异常处理的意义在于开发者能提供错误发生时要调用的代码。当程序出现错误时,Python 不再是简单终止,而是会搜索程序员提供的遇到相应问题时要执行的代码。如果找

到针对出现的错误要被调用的代码，就会执行被调用代码；如果找不到异常处理的代码，则会报错。

10.2 Python 的异常类

我们在之前看到的异常只是个名字，其实这些都是异常对象，即 Python 抛出的异常都是类。比如定义一个变量，然后以错误的名字调用：

```
>>>a = 100
>>>print(A)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print(A)
NameError: name 'A' is not defined
```

这可能是我们在学习变量时就遇到的异常，现在你看到最后一行的 NameError，应该能知道这是一个名字错误，具体的错误就是冒号后面说的 A 这个名字没有被定义，其实这就是程序引发了一个 NameError 异常。

像这样的内置异常类，Python 提供了很多，可以查看手册，表 10-1 是一些常见的异常类。

表 10-1　Python 常见异常类

异常类名	描　　述
Exception	所有异常的基类
NameError	尝试访问不存在的变量名
ImportError	导入模块引发的异常
SyntaxError	语法错误
IndexError	索引越界
IOError	I/O 错误，比如打开不存在的文件
KeyError	调用字典中不存在的键
AttributeError	尝试访问未知的对象属性
ValueError	函数传参类型不正确

知己知彼，百战不殆。熟悉了 Python 内置的异常类，才能更好地处理，当然，你也可以从实战中不断提升，也就是碰到异常再处理，不过，这样有些被动，最好的办法还是主动出击，提前设置异常处理。

图 10-1 是 Python 3.6 手册的异常树截图，Python 标准库(Library Reference)中找到 built-in Exception 这一节就能看到完整版。

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
```

图 10-1 内置异常类结构树

10.3 捕获和处理异常

异常分为两个阶段：引发异常的错误；检测并处理异常。异常处理的基本流程如下。

(1) 监测特定语句。

(2) 被检测语句如果出现错误或者发生异常，则中止执行错误代码。

(3) 抛出(引发)特定异常。

(4) 寻找捕获器来捕捉并处理相应的异常。

(5) 如果有捕获程序，则跳转到异常处理的代码；如果没有捕获程序，则直接由 Python 抛出异常并中止程序。

10.3.1 try...except...语句

Python 处理异常的基本语句是 try...except...语句，语句块要采用 4 格缩进的规范，Python 3 中的语法如下：

```
try:
    语句块      #被监控的语句，错误在此出现，Python 检查异常类型
except Exception["as" identifier]:          #捕获异常及异常信息
    语句块      #处理异常的语句，执行后会跳过其他异常块
except Exception["as" identifier]:
    语句块      #处理异常的语句
except Exception["as" identifier]:
    语句块      #处理异常的语句
主程序     #在 try...except...之后继续执行
```

注意：这个结构在 Python 2 中，except 语句为

```
except Exception,identifier:
```

下面通过 try...except...将 10.2 节变量不存在的例子中的异常处理掉。

```
#try_name.py
try:
    a = 100
    print(A)
except NameError:
    print("变量名不存在")
```

输出结果:

```
>>>变量名不存在
```

这里我们将可能出错的代码放到 try 的代码块中执行。如果没有异常就正常执行完毕,不会执行到 except 的代码;如果有异常抛出,则执行 except 的代码,except 后面跟的就是你要捕捉的异常类的名字,如果这个异常类不是上面 try 语句所抛出的,就意味着与没有做异常处理一样,你该看见的报错还是会出现,程序也会中止。这个例子中抛出和捕获的异常是对应的,在接收到异常之后就会执行 except 语句的代码块,在这里是为了给用户一个友好的中文反馈。

10.3.2 try...except...else 语句

如果有多个异常需要处理可以通过罗列多个 except 语句实现,如果 try 的代码块没有任何异常抛出,还可以利用 else 执行一段代码。在 if、for、while 等语句中已经见过 else,异常处理的 else 语句的代码块是在 try 语句代码块没有异常时执行,下面举例说明一下多个 except 加 else。

```
#try_except_n.py
try:
    x = input("被除数:")
    y = input("除数:")
    z = int(x) / int(y)
except ValueError:
    print("请输入数字!")
except ZeroDivisionError:
    print("除数不能为 0")
else:
    print(x, " / ",  y, " = ", z)
```

如果输入的值不是数字就会抛出 ValueError 异常,处理后输出提示信息,不执行 else。

```
被除数:a
除数:a
请输入数字!
```

如果除数是 0，效果如下。

```
被除数:10
除数:0
除数不能为 0
```

没有异常就会执行 else 语句的代码块，效果如下。

```
被除数:10
除数:2
10 / 2 =  5.0
```

10.3.3 finally 子句以及嵌套

finally 语句的代码块，不管有没有异常都会执行。finally 通常会和 try...except...组合使用，那有什么是必须要执行的呢？很多需要我们强制关闭的对象都需要，比如对一个文件操作，无论是否有异常，最后都要关闭文件，这种情况就适用于 finally，我们来看一个例子。

```
#try_file.py
try:
    f = open("test.txt", 'r')
    print(f.read())
except IOError:
    print("文件不存在")
finally:
    #f.close()
    try:
        f.close()
    except NameError:
        pass
```

在 try 中读取文件，可能会出现要打开的文件不存在的问题，这就会引发 IOError，将其捕获即可，文件对象用完之后要关闭，在 finally 子句中可以直接将 f.close()关掉，但是这行代码还可能产生异常。如果前面打开文件失败，就不会有 f 对象，那么 f.close()就会报 NameError 异常，既然会有异常，那么就再次嵌套一个 try 即可。

10.3.4 谁都跑不了

在异常手册中可以看到，异常的种类很多，我们不可能把所有的异常一一捕获。你还会发现，所有异常的最顶层其实是 BaseException 类，所以，有个终极大招就是直接捕获 BaseException 类，这样，所有的异常都跑不掉了。

现在就写一个函数，把参数转为整数。

```
#try_int
def toInt(x):
    try:
```

```
        i = int(x)
    except BaseException as err:
        i = err
    return i

print(toInt('abc'))
print(toInt("123"))
print(toInt([456]))
```

每个错误都不同，所以捕获的异常也都不同，结果如下：

```
invalid literal for int() with base 10: 'abc'
123
int() argument must be a string, a bytes- like object or a number, not 'list'
```

10.4 抛出异常

现在了解 Python 会在发生错误时自动抛出异常，我们也可以自己主动抛出异常，除了内置异常外，还可以自定义异常类。主动抛出异常不是为了制造麻烦，而是为了引导程序向正确的方向运行，比如要检查用户输入的数据，如果不符合要求就主动引发异常，并向异常传递数据。

10.4.1 raise 语句

raise 语句是 Python 提供的一种自行引发异常的机制。我们可以在 raise 语句中指定异常类型以及附加的异常信息。raise 语句的常用语法如下：

```
raise 异常类名(附加异常信息)
```

比如要检测一个字符串长度，如果长度大于 5 则抛出异常，并且配合 try 来捕获我们自己抛出的异常。

```
#raise_exception
s = "123456"
try:
    if len(s) > 5:
        #raise Exception
        raise Exception("超过 5 个字符")
except Exception as err:
    print(err)
```

符合 if 条件，由 raise 抛出一个 exception 异常以及括号中的提示信息，这样捕捉到异常后就可以得到这个信息，当然，也可以没有提示信息，就像注释行那样，结果如下。

```
超过 5 个字符
```



10.4.2　自定义异常类

如果内置异常类中没有符合程序要抛出的异常类型，我们也可以自定义异常类。在 Python 中，可以通过继承 Exception 类来创建自己的异常类，而且通常只需要定义几个属性就可以了。下面就来自定义一个异常类，并在需要时抛出。

```
#my_error_class.py
class MyError(Exception):
    def __init__(self, length, least):
        self.length = length
        self.least = least

if __name__ == "__main__":
    try:
        s = input("请输入一个字符串:")
        if len(s) > 5:
            raise MyError(len(s), 5)     #抛出自定义异常并传参
    except MyError as m:
        print("MyError: 输入长度为 %s,长度不能超过 %d" % (m.length, m.least))
```

这个异常类定义好之后，跟内置类的用法一样，在需要时 raise 出来就行了，结合面向对象的知识再反观异常类，抛出来的其实就是一个对象，通过调用对象的属性和方法来获取更多的信息，结果如下：

```
>>>
请输入一个字符串:abc
>>>
请输入一个字符串:abcdefg
MyError: 输入长度为 7,长度不能超过 5
```

10.4.3　assert 语句

assert 语句同样可以抛出异常，主要用在判断表达式中，表达式为真时，什么都不做；表达式为假时，则抛出异常。例如：

```
>>>assert "abc" == "abc"
>>>assert "abc" == "abcdef"
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    assert "abc" == "abcdef"
AssertionError
```

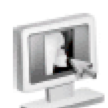
可以看到，表达式为假时直接抛出了一个 AssertionError 异常，也可以把 assert 看作是简化的 raise，其实看起来就像 if 和 raise 结合的结果。当然，这个异常也是可以捕获的。

```
>>>try:
```

```
    assert "abc" == "abcdef"
except AssertionError:
    print("Error")
```

输出结果：

```
Error
```



重点提示

在这一章中，你要掌握的内容：

（1）知道常见异常代表的意义。

（2）能够捕获异常并处理。

（3）掌握异常的几个语句。

（4）会自定义异常类。



动动手

（1）编写程序实现用户输入两个值，由程序执行除法，通过异常处理非数字的错误、除数为0的错误以及运行过程中可能出现的其他错误，如果不确定异常名字，可以先试错。

（2）系统提供的内建函数open()，在打开文件时不能处理异常，请改进open()并封装一个新的函数，当程序刚打开文件时直接返回句柄，如果发生异常则返回None，并且不报异常。

第 11 章 开发图形用户界面

现在，我们所有涉及与用户交互的程序都是在 IDLE 或者终端中的命令行模式，即是纯文本的。对于学习来说这已经足够了，但是如果想要让程序对用户来说更友好，这显然是不够的。

因为对于用户来说，在生活中能涉及的与程序互动的方式，大多数都是图形化的，比如计算机上安装的各种各样的软件、手机上的应用程序等，全部都是图形化的。

所以我们现在来了解一下如何用 Python 开发图形化的用户界面，在这一章里我们会设计一些简单的 GUI。还是那句话，这是一个入门的过程，不是一个字典似的资料库，我们要做的是掌握方法。就像拼积木，我们现在要掌握怎么搭一个小房子，你只要会搭小房子，就可以一点一点搭起一个高楼了。

11.1 GUI

GUI 是图形用户界面(Graphical User Interface)的简称，又称图形用户接口，是指采用图形方式显示的计算机操作用户界面。用户可以通过 GUI 与程序进行交互，这样做的好处是减少用户的学习成本，稍有 GUI 使用经验的用户都可以快速上手一个新的程序。

Python 有非常丰富的 GUI 框架，Python wiki GUI programming 给出了超过 30 个跨平台框架方案，包括 Pyjamas 这样的跨浏览器 Web 开发框架。下面介绍其中的几个。

1. tkinter

tkinter(也叫 Tk 接口)是 Python 内置的标准 GUI 库。tkinter 是一个轻量级的跨平台图形用户界面(GUI)开发工具，可以运行在大多数的 UNIX 平台、Windows 和 Macintosh 系统。tkinter 用起来也非常简单，Python 自带的 IDLE 就是 tkinter 实现的，包括我们在前面用简括 turtle 绘图、画正弦波等图形的呈现界面也都是 tkinter 实现的。

2. wxPython

wxPython 是 Python 语言实现的一套优秀的 GUI 图形库，允许 Python 程序员很方便地创建完整的、功能键全的 GUI。wxPython 是作为优秀的跨平台 GUI 库 wxWidgets 的 Python 封装和 Python 模块的方式提供给用户的。其功能要强于 tkinter，相比较来说更适合开发大型的 GUI 应用。

3. Qt

Qt 是一个 C++ 编写的跨平台开发框架，如果你的应用是完全开源的，就可以免费使用 Qt，否则你需要购买商业许可。Qt 已经存在很久，一度属于诺基亚公司，作为一个非常全面的工具代码库和 API，被大量行业广泛采用，覆盖包括移动在内的多个平台。卫星导航应用的图形用户界面往往就是 Qt 开发的。

4. Kivy

Kivy 是一个非常有趣的项目，它基于 OpenGL ES 2，支持 Android 和 iOS 平台的原生多点触摸，作为事件驱动的框架，Kivy 非常适合游戏开发和处理从 widgets 到动画的任务。如果你想开发跨平台的图形应用，或者仅仅需要一个强大的跨平台图形用户开发框架，Kivy 是不错的选择。

如果对 GUI 要求不是很高，可以直接用 tkinter。学习使用一个新的框架不像学习一门计算机语言那么难，学习 GUI 编程，并不是在学习 Python 语言，而是在学习一个计算机工具怎么使用。

11.2 tkinter

作为内置标准库的 tkinter，用起来很简单，但是文档和功能上相比其他第三方框架要差很多，不过基本的 GUI 开发还是没问题的，我们通过 tkinter 来做个热身，熟悉一下 GUI 开发吧。

用 tkinter 开发 GUI，可以先想象一下在底板上拼图，要先拿一块空白底板，然后拿现成的组件摆在上面组成完整的布局。


tkinter

11.2.1 创建空白窗口

用 tkinter 开发 GUI 需要先导入 tkinter 模块，这个模块不需要安装，直接导入即可，可以在交互模式下试一下：

```
>>>import tkinter
```

如果没有异常就表示模块可用。

导入模块后，第一步要做的是生成一个主窗口对象（想象那个空白底板）。第二步是通过 tkinter 模块的其他函数、方法等对主窗口对象添加组件。创建主窗口只需要 3 行代码。

```
>>>import tkinter
>>>root = tkinter.Tk()
>>>root.mainloop()
```

其中，第一行导入模块，第二行实例化主窗口对象，第三行进入消息循环。这时就可以在屏幕上看到图 11-1 所示的一个空白窗口了。

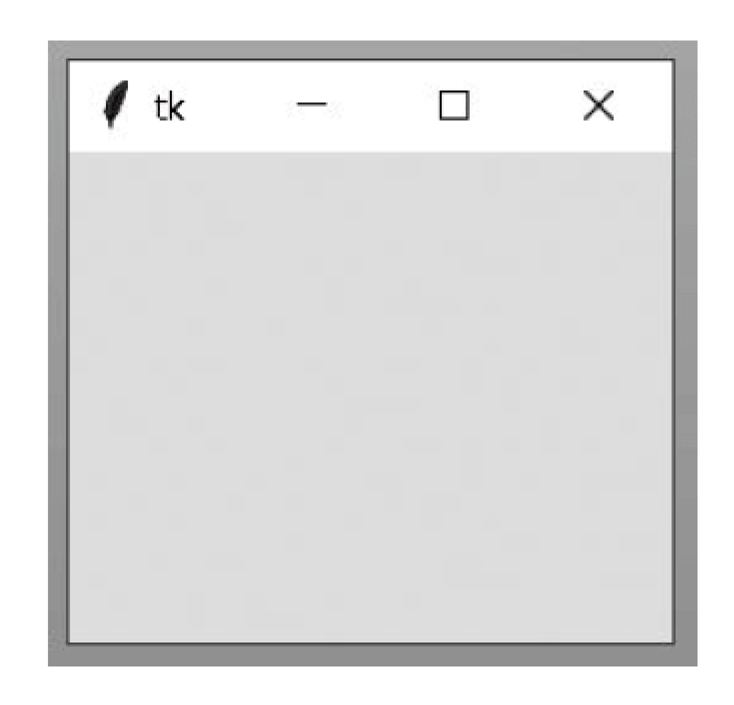


图 11-1 tkinter 空白窗口

前两步比较好理解，第三步消息循环简单说就是用来处理窗口及内部组件的事件的。鼠标的移动、键盘的按键、单击按钮这些输入叫作事件。比如鼠标移动事件、单击事件、回车按下事件等。图形操作都是持续的、动态的，像动画一样。

mainloop 进入事件（消息）循环。这一步会把控制权交给 GUI，一旦检测到事件，就刷新组件。比如输入一个字符，就要立即在光标位置显示出来（前提是你选中了文本框，也就是鼠标在文本框这个图案的范围内单击过）；又比如单击了一个图形界面里的按钮，就画一个五角星。

11.2.2 添加组件

文本框、按钮、标签等都称为组件（widget），有了空白窗口就可以向窗口添加组件了，比如文字标签、按钮。添加组件需要先生成组件对象，再通过 pack 方法添加至主窗口。

```
#tk1.py
import tkinter
root = tkinter.Tk()       #生成主窗口
label = tkinter.Label(root, text= "你好")     #生成标签对象
label.pack()              #将标签添加到主窗口
button1 = tkinter.Button(root, text= "五角星")     #生成按钮
button1.pack()            #将按钮添加到主窗口
root.mainloop()
```

运行后可以看到图 11-2 所示的窗口。



图 11-2 窗口组件

这里我们添加了两个组件，一个标签“你好”，一个按钮“五角星”，因为没有指定组件的位置，所以 tkinter 就自行安排了。

11.2.3 事件绑定

11.2.2 小节的例子如果你写出来,可能会发现,单击“五角星”按钮没有任何反应。这是因为我们并没有为单击“五角星”按钮这个行为编写相关联的程序。

可以针对具体事件编写对应的处理函数,并通过 bind 方法与组件进行绑定。这样,当组件触发响应事件后就会调用这个函数,现在就给“五角星”按钮绑定一个处理函数。

```
#tkf.py
import tkinter
import turtle
root = tkinter.Tk()                              #生成主窗口
label = tkinter.Label(root,text= "你好")         #生成标签
label.pack()                                     #将标签添加至主窗口
button1 = tkinter.Button(root,text= "五角星")    #生成按钮
button1.pack()                                   #将按钮添加至主窗口
####
def wjx(event):                                  #事件响应函数
    for i in range(5):
        turtle.forward(100)
        turtle.right(144)
####
button1.bind('<Button-1>',wjx)                   #绑定事件
root.mainloop()                                  #进入消息循环
```

现在再单击“五角星”按钮,就会启动海龟绘图画五角星了。

11.2.4 其他组件

tkinter 提供的组件很丰富,目前有十几种 tkinter 组件,列举如下。

Button:按钮控件,在程序中显示按钮。

Canvas:画布控件,显示图形元素如线条或文本。

Checkbutton:多选框控件,用于在程序中提供多项选择框。

Entry:输入控件,用于显示简单的文本内容。

Frame:框架控件,在屏幕上显示一个矩形区域,多用来作为容器。

Label:标签控件,可以显示文本和位图。

Listbox:列表框控件,在 Listbox 窗口小部件用来显示一个字符串列表给用户。

Menubutton:菜单按钮控件,用于显示菜单项。

Menu:菜单控件,显示菜单栏、下拉菜单和弹出菜单。

Message:消息控件,用来显示多行文本,与 Label 比较类似。

Radiobutton:单选按钮控件,显示一个单选的按钮状态。

Scale:范围控件,显示一个数值刻度,为输出限定数字区间。

Scrollbar:滚动条控件,当内容超过可视化区域时使用,如列表框。

Text:文本控件,用于显示多行文本。

Toplevel:容器控件,用来提供一个单独的对话框,与 Frame 比较类似。

Spinbox：输入控件，与 Entry 类似，但是可以指定输入范围值。

PanedWindow：是一个窗口布局管理的插件，可以包含一个或者多个子控件。

LabelFrame：是一个简单的容器控件，常用于复杂的窗口布局。

tkMessageBox：用于显示应用程序的消息框。

11.2.3 小节的程序再加上一个菜单，程序如下。

```
#tkfm.py
import tkinter
import turtle
root = tkinter.Tk()                                        #生成主窗口
####
menu = tkinter.Menu(root)                                  #生成菜单
submenu = tkinter.Menu(menu,tearoff = 0)                   #生成下拉菜单
submenu.add_command(label = "Open")                        #向下拉菜单添加指令名字为 Open
submenu.add_command(label = "Save")                        #向下拉菜单添加指令名字为 Save
menu.add_cascade(label = "File",menu = submenu)            #将下拉菜单添加到菜单
root.config(menu = menu)
####
label = tkinter.Label(root,text = "你好")                  #生成标签
label.pack()                                               #将标签添加至主窗口
button1 = tkinter.Button(root,text= "五角星")              #生成按钮
button1.pack()                                             #将按钮添加至主窗口
####
def wjx(event):                                            #事件响应函数
    for i in range(5):
        turtle.forward(100)
        turtle.right(144)
####
button1.bind('<Button-1>',wjx)                             #绑定事件
root.mainloop()                                            #进入消息循环
```

运行程序可以看到如图 11-3 所示的下拉菜单了，但是并没有绑定事件。

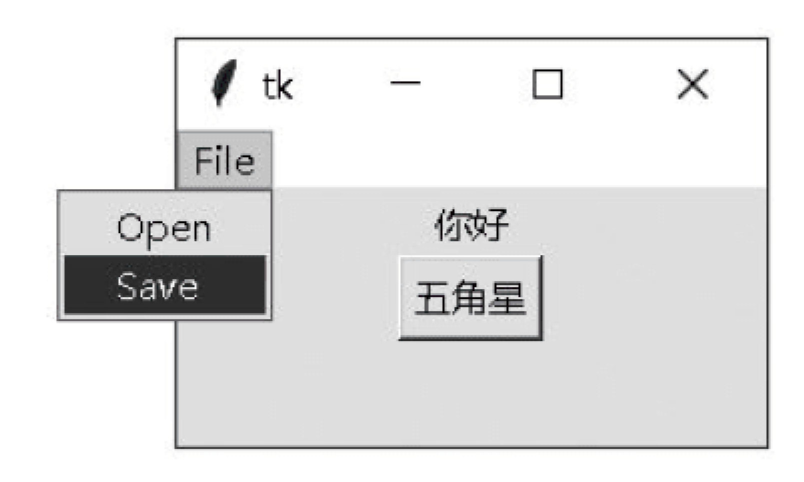


图 11-3　下拉菜单

11.3 wxPython

虽然 tkinter 用起来很方便，不过只适合开发小型应用程序，若要开发复杂的应用程序就不够用了，这时就需要考虑其他的 GUI 库。11.1.1 小节我们也介绍了一些，这里以

wxPython 为例。

wxPython 不是 Python 内置模块，所以需要额外安装（见图 11-4），最简单的方法就是通过 pip 安装，“-U”是表示升级，如果有新版本会自动更新。注意 wxPython 中的字母“P”是大写的。

```
Windows PowerShell
PS C:\Users\milo> pip install -U wxPython
Collecting wxPython
  Downloading https://files.pythonhosted.org/packages/de/78/bbd1e102c25e5f38aae4040c4bf5ec
thon-4.0.1-cp36-cp36m-win32.whl (14.1MB)
    100% |████████████████████████████████| 14.1MB 48kB/s
Collecting six (from wxPython)
  Downloading https://files.pythonhosted.org/packages/67/4b/141a581104b1f6397bfa78ac9d43d8
1.11.0-py2.py3-none-any.whl
Installing collected packages: six, wxPython
  Found existing installation: six 1.10.0
    Uninstalling six-1.10.0:
      Successfully uninstalled six-1.10.0
  Found existing installation: wxPython 4.0.0b2
    Uninstalling wxPython-4.0.0b2:
      Successfully uninstalled wxPython-4.0.0b2
Successfully installed six-1.11.0 wxPython-4.0.1
```

图 11-4　安装 wxPython

此外，也可以登录 wxPython 的官方网站下载安装，而且官方网站还有帮助和手册可以参考，地址是 https://www.wxpython.org/。

安装后可以导入 wx 模块测试一下，没有报错就可以使用了。

```
>>>import wx
```

11.3.1　子类化开发：空白窗口

在 tkinter 的例子中，如果细心观察可以发现，我们做的很多事情都是在对类实例化，再通过对象调用方法的形式进行程序设计。这种做法比较适合简单窗口的开发，如果是复杂窗口，可能要实例化很多对象，开发起来就比较麻烦。



子类化开发：空窗口

所以，还有一种做法就是通过继承相应的类并进行重载，把程序设计都放在子类中，这样的程序模块化程度更高，也更灵活，这种方式可以称为子类化开发，在 wxPython 中可以采取这种方式，当然，tkinter 也是可以采用的。

通过一个空白窗口进入子类化开发模式。GUI 的开发大同小异，用 wxPython 生成空白窗口只需要以下 5 个步骤。

（1）导入 wx 模块。

（2）实例化一个应用程序对象。

（3）创建 wx.Frame 主窗口对象，其他组件可以加入 wx.Frame 窗口中。

（4）通过 Frame 的 show()方法显示窗体。

（5）进入应用程序事件主循环。

我们先采用对象化的方式生成一个空白窗口，这样，每个对象起什么作用就比较明显了。

```
#wxpy1.py                    #对象化生成空白窗口
```



```
import wx
app = wx.App()                          #应用程序对象
frame = wx.Frame(parent = None)         #框架窗口对象
frame.Show()                            #显示框架窗口
app.MainLoop()                          #进入消息循环
```

运行后就可以看到一个空白窗口了(见图 11-5)。



图 11-5　wxPython 空白窗口

这里可以看到 wx. App、wx. Frame 其实都是类,我们可以继承过来进行重载,最后再实例化就可以了。下面把工作简化,继承 wx. App 类进行重载。

```
#wxpyc1.py
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None)     #生成框架窗口
        frame.Show()                        #显示框架窗口
        return True
app = MyApp()
app.MainLoop()
```

可以看出新定义一个子类继承 wx. App,在这个类里实例化了 wx. Frame。也可以重新加载 wx. Frame,我们先把工作集中在一个点上。

11.3.2　添加组件及窗口布局

有了空白窗口,就可以添加组件了。这里有个关键角色就是 wx. Frame, wx. Frame 是继承自 wx. Window 类的窗口类,窗口可以指定大小和位置,可以加入标题、标签、菜单、工具栏、状态栏等组件。



组件及布局

先对聊天窗口添加组件,并调整好组件的布局。

```
#wxmychat-0.1.py
import wx

class MyApp(wx.App):
    def OnInit(self):               #重载 OnInit 方法
        #生成 frame 框架对象
```

```
        frame = wx.Frame(parent = None,
            title = "Milo Chat",               #窗口标题
            size = (520,450),                  #窗口尺寸
            pos = (800,300))                   #窗口启动时在屏幕的位置

        panel = wx.Panel(frame,-1)             #生成面板,以便布局管理
        labelall = wx.StaticText(panel,-1,     #标签
            'All Contents',                    #标签内容
            pos = (210,5))
        self.textall = wx.TextCtrl(panel,-1, #文本框
            size = (480,200),
            pos = (10,30),
            style = wx.TE_READONLY|wx.TE_MULTILINE)

        labelin = wx.StaticText(panel,-1,'I Say',
            pos = (230,230))
        self.textin = wx.TextCtrl(panel,-1,
            size = (480,100),
            pos = (10,260),
            style = wx.TE_MULTILINE)

        self.buttonSent = wx.Button(panel,-1,"Sent",   #按钮上的文字
            size = (75,25),pos = (175,370))
        self.buttonSave = wx.Button(panel,-1,"Save",
            size = (75,25),pos = (260,370))

        frame.Show()
        return True

if __name__ == "__main__":
    app = MyApp()
    app.MainLoop()
```

程序运行后的效果如图 11-6 所示。

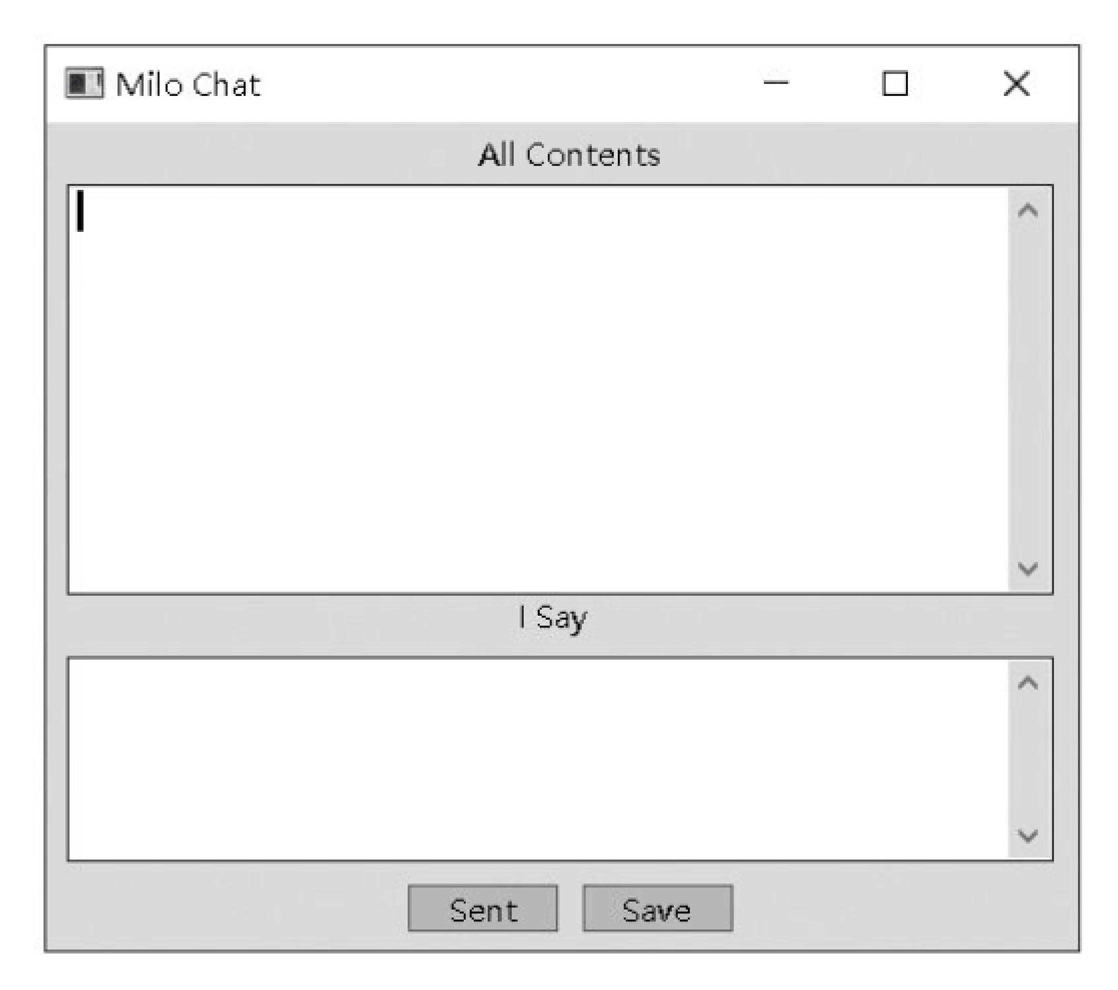


图 11-6　聊天窗口

这段代码看起来好像很多，其实只做了两件事，就是添加组件和布局管理，我们来逐一介绍一下。

1. 框架和面板

通过 wx.Frame 生成了框架，框架就是我们说的窗体，可以在里面添加组件。wx.Frame 是所有框架的父类，构造函数 wx.Frame.__init__的作用主要是针对框架的一些基本属性设置，可以在子类中直接调用，也可以像现在这样实例化出来。

```
wx.Frame.__init__(parent,id,title,pos,size,style,name)
```

参数说明如下。

parent：框架父窗体，作为顶级窗体，值为 None。

id：定义新窗体的 ID 号，如果不指定可以用－1。

title：窗体标题。

pos：指定窗体左上角在屏幕中的位置，(0,0)代表显示器左上角。例子中(800,300)表示距左侧 800 像素，距顶部 300 像素。

size：当前窗体的初始尺寸。

style：窗体的类型。

name：框架名字，如果指定，可以通过名字找到这个窗体。

这个例子中还有一个 wx.Panel(面板)，面板的作用是管理组件的布局。可以想象是在木框架上贴了一张画布，在画布上画画。其参数同 wx.Frame.__init__，因为此例已经在 Frame 中全部指定，所以除了 Parent 以外，其他都省略了。

2. 组件

有了框架和面板，接下来就是添加组件了，依次添加了三种组件分别如下。

wx.StaticText：静态文本框。

wx.TextCtrl：文本框。

wx.Button：按钮。

每个组件都有自己特定的属性，具体可以通过 wxPython 的官方文档查阅。这里所有组件的第一个参数都是 parent，其实就是组件添加的面板；组件的 pos 是指组件左上角位于面板中的位置，(0,0)是面板的左上角；size 是组件的大小，比如按钮和文本框都指定了大小。

这里的两个文本框都用 style 参数来指定文本框的样式，这个 style 非常有用，可以用来指定文本框是否支持多行、是否可拉伸等。比如 wx.TE_MULTILINE 就表示这是一个支持多行的文本框，如果没有这条，那么文本框只能输入或显示一行文本。self.textall 对象的 style 除了多行属性还指定了 wx.TE_READONLY(文本框只读)，需要注意的是，如果是多个属性，需要用“|”隔开。

11.3.3 事件绑定

事件绑定就是在窗口内进行操作时触发的函数，设计的窗口只有两个按钮：发送消息按钮和清空聊天记录按钮。因为现在还没有学习如何通过网络传输数据，所以简化一下这个例子，自己和自己聊天：在下面输入消息，单击 sent 按钮后将信息显示到上面的聊天记录窗口，单击 clear 按钮，则清空聊天记录。


文本框和聊天窗口

函数设计及绑定方式如下：

```
#wxmychat-0.2.py
import wx
import time

class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None,
                         title = "Milo Chat",
                         size = (520,450),
                         pos = (800,300))
        panel = wx.Panel(frame,-1)
        labelAll = wx.StaticText(panel,-1,'All Contents',
                                 pos = (210,5))
        self.textAll = wx.TextCtrl(panel,-1,
                                   size = (480,200),
                                   pos = (10,30),
                                   style = wx.TE_READONLY|wx.TE_MULTILINE)
        labelIn = wx.StaticText(panel,-1,'I Say',
                                pos = (230,230))
        self.textIn = wx.TextCtrl(panel,-1,
                                  size = (480,100),
                                  pos = (10,260),
                                  style = wx.TE_MULTILINE)
        #############################################################
        self.buttonSent = wx.Button(panel,-1,"Sent",
                                    size = (75,25),pos = (175,370))
        self.Bind(wx.EVT_BUTTON,
                  self.OnButtonSent,self.buttonSent)
        self.buttonClear = wx.Button(panel,-1,"Clear",
                                     size = (75,25),pos = (260,370))
        self.Bind(wx.EVT_BUTTON,
                  self.OnButtonClear,self.buttonClear)
        #############################################################
        frame.Show()
        return True

    def OnButtonSent(self,event):
        userinput = self.textin.GetValue()      #获取输入文本框的数据
        self.textIn.Clear()                     #清空输入文本框
```

```
        nowtime = time.ctime()                    #增加时间
        inmsg = "You (%s) :\n%s \n" % (nowtime,userinput)    #拼接数据
        self.textAll.AppendText(inmsg)      #尾部添加文本

    def OnButtonClear(self,event):
        self.textAll.Clear()

if __name__ == "__main__":
    app = MyApp()
    app.MainLoop()
```

事件绑定通过 wx. App. Bind，用到了其中三个参数，Bind（事件类型、事件响应函数、事件源）。

(1) 事件类型：几乎涵盖了所有窗口操作，我们用的是按钮事件 wx. EVT_BUTTON。

(2) 事件响应函数：响应函数除了 self 参数外，还需要第二个参数接受一个 event 对象，不同的事件会接受不同的 event 对象。

(3) 事件源：对应的是具体组件，这里对应的就是按钮。

定义的事件响应函数要根据实际逻辑关系设计，比如 sent 按钮绑定的 OnButtonSent。通过调用输入文本框对象 self. textin 的 GetValue 方法，获取输入文本框内的数据；再调用 Clear 方法清空输入文本框，从视觉上好像消息发走了；然后再将获取的数据进行处理，比如加上发送的时间；最后这个消息输出到代表聊天记录的文本框，而且不能覆盖之前的消息，通过代表聊天记录的对象 self. textall 调用 AppendText 方法实现。

最后的效果如图 11-7 所示。

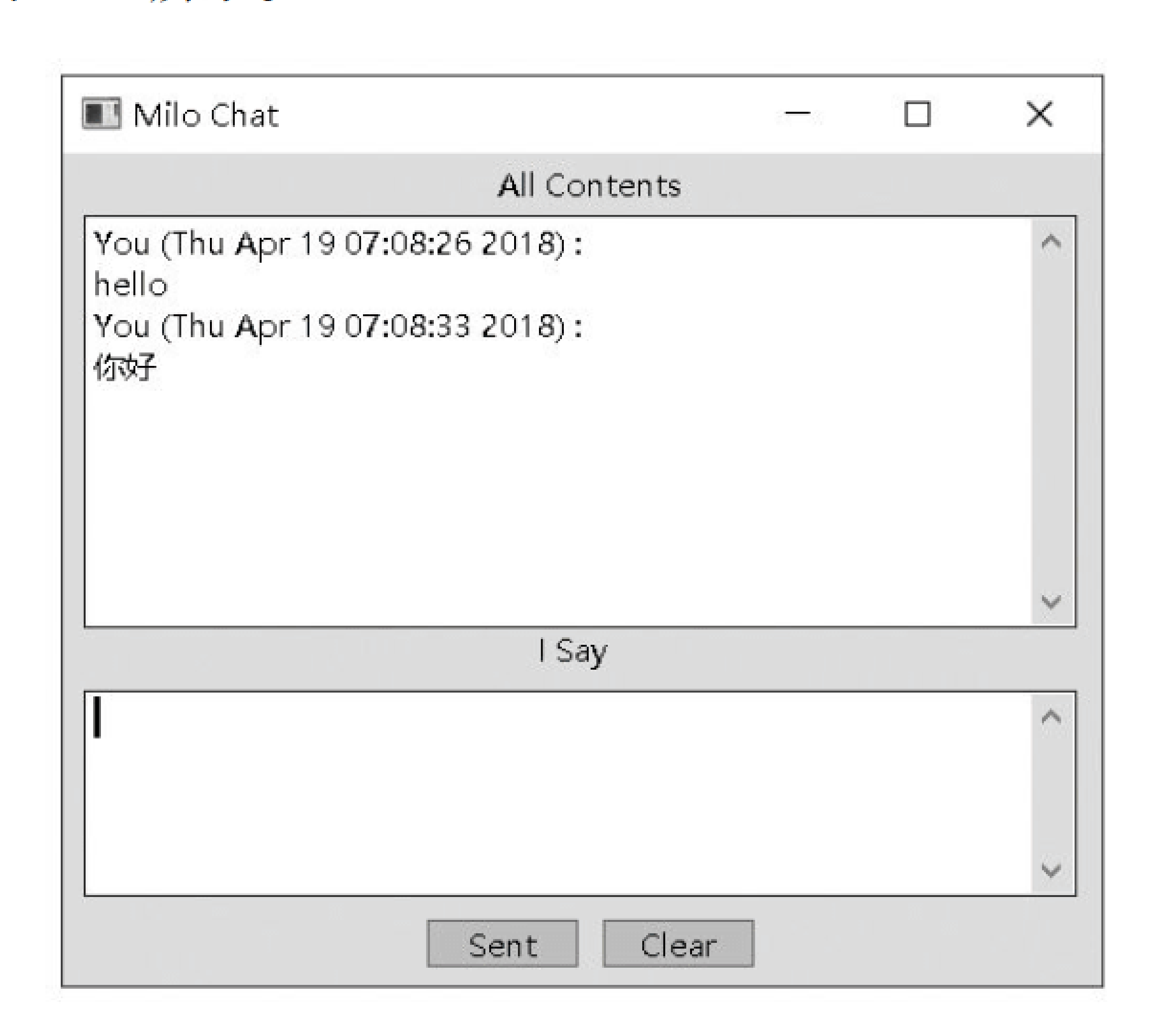


图 11-7 聊天记录

这样就完成了一个简单的聊天窗口，配合网络编程就可以传输数据了。

11.3.4 布局管理器


布局管理器

我们完成了一个聊天窗口，但是这个窗口还有一些问题，比如如果拖拽、缩放、拉伸它（见图 11-8），内部组件并不会随着边框按比例缩放，这是因为在定义组件时，用了固定尺寸。这种固定尺寸的形式对于像计算器那样大小固定的软件比较有用，但在这里显然不太合适，这时我们就要了解布局管理器了。

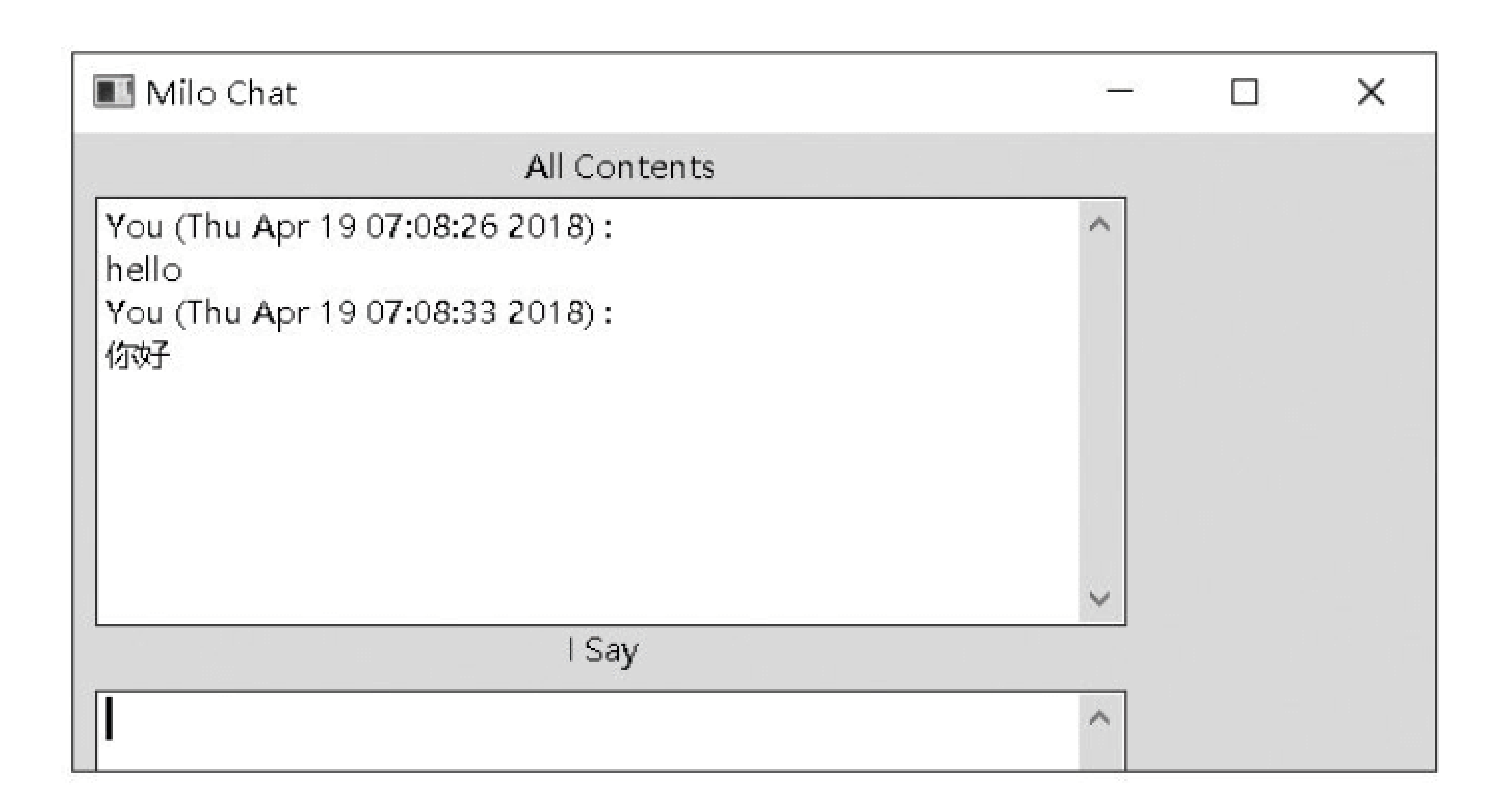


图 11-8 聊天窗口拉伸

wxPython 中，使用 wx.BoxSizer 管理组件的布局。wx.BoxSizer 将窗口中的组件以单元格的形式划分，将组件放至单元格中，而不需要单独规定大小，仅仅通过 sizer 就可以管理添加在其中的组件布局位置。

sizer 的创建和使用的步骤如下。

（1）创建 sizer 对象：

```
sizer = wx.BoxSizer(布局方式)
```

（2）将容器中的子窗口添加到这个 sizer：

```
sizer.Add(组件对象,proportion,flag,border)
```

（3）将 sizer 关联到一个容器：

```
panel.SetSizer(sizer)
```

以这个聊天窗口为例，可以这样划分（见图 11-9）。

整体为纵向排列，其中最后一行为横向排列，利用 sizer 中的纵向和横向划分就可以了，代码如下：

```
#wxmychat-0.3.py
import wx
import time
```





图 11-9　布局划分

```
class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self,None,-1,"Milo Chat",size = (520,450))
        panel = wx.Panel(self)
        labelAll = wx.StaticText(panel,-1,'All Contents')
        self.textAll = wx.TextCtrl(panel,-1,
                                size = (480,200),
                                style = wx.TE_READONLY|wx.TE_MULTILINE)
        labelIn = wx.StaticText(panel,-1,'I Say')
        self.textIn = wx.TextCtrl(panel,-1,
                               size = (480,100),
                               style = wx.TE_MULTILINE)
        self.btnSent = wx.Button(panel,-1,"Sent",size = (75,25))
        self.btnClear = wx.Button(panel,-1,"Clear",size = (75,25))
        self.Bind(wx.EVT_BUTTON,self.OnButtonSent,self.btnSent)
        self.Bind(wx.EVT_BUTTON,self.OnButtonClear,self.btnClear)
        ###布局管理 ####
        btnSizer = wx.BoxSizer()                    #创建横向管理器
        btnSizer.Add(self.btnSent,proportion = 0)
        btnSizer.Add(self.btnClear,proportion = 0)

        mainSizer = wx.BoxSizer(wx.VERTICAL)    #创建纵向管理器
        mainSizer.Add(labelAll,proportion = 0,flag = wx.ALIGN_CENTER)
        mainSizer.Add(self.textAll,proportion = 1,flag = wx.EXPAND)
        mainSizer.Add(labelIn,proportion = 0,flag = wx.ALIGN_CENTER)
        mainSizer.Add(self.textIn,proportion = 0,flag = wx.EXPAND)
        mainSizer.Add(btnSizer,proportion = 0,flag = wx.ALIGN_CENTER)

        panel.SetSizer(mainSizer)
        mainSizer.SetSizeHints(self)                #启动组件最小限制
        ################

    def OnButtonSent(self,event):
```

```
        userinput = self.textIn.GetValue()
        self.textIn.Clear()
        nowtime = time.ctime()
        inmsg = "You (%s) :\n%s \n" % (nowtime,userinput)
        self.textAll.AppendText(inmsg)    #尾部添加文本

    def OnButtonClear(self,event):
        self.textAll.Clear()

if __name__ == "__main__":
    app = wx.App()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

运行后,可以随边框拉伸,动态效果你可以在自己的计算机上尝试(见 11-10)。

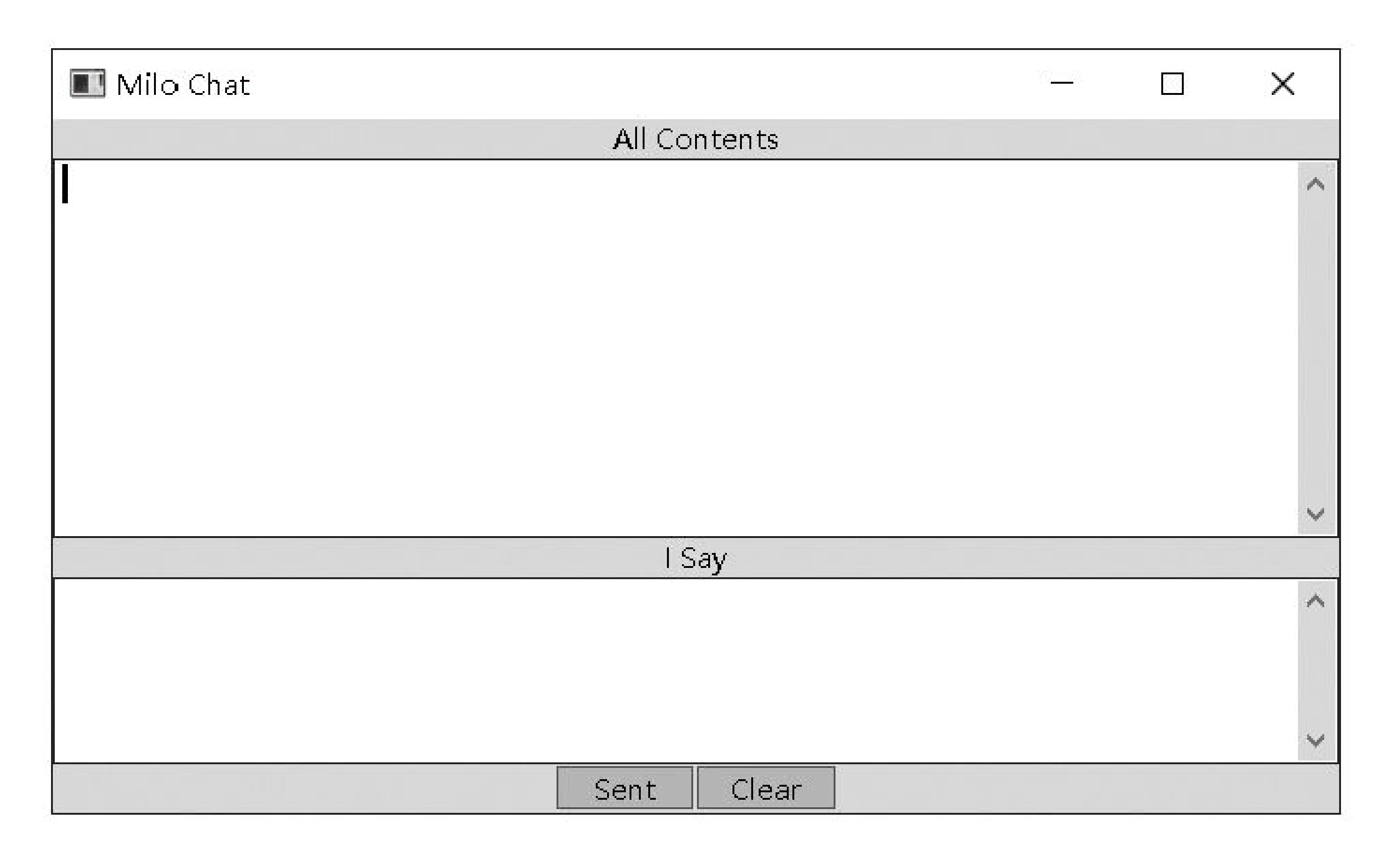


图 11-10　窗口拉伸

在这段代码中我们做了一些更改,针对这段代码说明如下。

(1) 换了个继承的类。因为工作基本是针对框架的,所以直接继承 wx.Frame 类进行开发。当然,用之前的办法也是可以的,这样做只是想在有限的内容编排下多展示一些开发的方式。

(2) 所有用来指定组件位置(pos)的参数都没有了。其中,框架、文本框和按钮都指定了 size,但由于组件是采用 sizer 管理的,所以,这里指定的 size 是窗口和文本框的初始尺寸。

(3) #号标注的部分就是布局管理器,根据前面布局分析,将整个窗口看作是纵向排列的,其中最后一排是横向的。所以我们创建两个布局,一个横向的 btnSizer,添加了两个组件,这两个组件在这个管理器中会以横向的方式排列。再创建一个纵向的 mainSizer,加入其中的组件就会按纵向排列,依次添加组件,最后将横向管理器作为一个整体添加进去。

(4) 向布局管理器添加组件用 sizer. Add(组件对象,proportion,flag)方法,其中 proportion 参数定义了构件在既定方向上所占空间的比例,是相对的,相对于其他组件。0 表示不变,比如按钮在拉伸时不需要变大。输入文本框为 0,则在其布局管理器的方向拉

伸时不会有变化，本例中上、下拉伸时输入文本框就不会纵向拉伸。而聊天记录文本框的值为 1，则在纵向拉伸时会随着变化。flag 则在布局的控制上加入了更多可能，本例中用了 wx. ALIGN_CENTER(居中)和 wx. EXPAND(组件使用所有分配给它的空间)。

(5) panel. SetSizer(mainSizer)用来设置面板采用哪个布局管理器。因为我们把按钮的 sizer 添加到了 mainSizer 中，所以，只需绑定 mainSizer 就可以了。

(6) 最后 mainSizer. SetSizeHints(self)的作用是防止窗口过度缩小，启动组件最小限制为初始的 size 值。

wxPython 提供了非常丰富的布局，可以用来制作复杂的界面。万事开头难，掌握了基本原理，你就可以通过手册或者相关资料进行复杂界面的开发了。推荐参考《wxPython in Action》中文版。

11.4 GUI 可视化构建工具：用 wxFormBuilder 开发 GUI 程序

掌握了 GUI 编程的基本原理，你可能会发现，如果要开发一个复杂的界面就意味着要写大量的代码。手动用代码创建好看的 GUI 是很痛苦的，一个可视化 GUI 工具会帮你减少这种痛苦。支持 wxPython 的 GUI 工具有很多：wxFormBuilder；wxDesigner；wxGlade；BoaConstructor；gui2py。

但是有些版本已经很久不更新了，也不支持 Python 3，如果有需要，最好选择一个仍然持续更新的版本，下面我们以 wxFormBuilder 为例介绍可视化工具。

wxFormBuilder 是一个支持多种程序语言的 GUI 构建工具，下载地址为 https://sourceforge. net/projects/wxformbuilder/? source = typ_redirect，最近的版本是 2016 年 9 月 23 日更新的。这里我下载的是 Windows 的. exe 版，正常安装就可以了。

启动后的软件界面如图 11-11 所示。

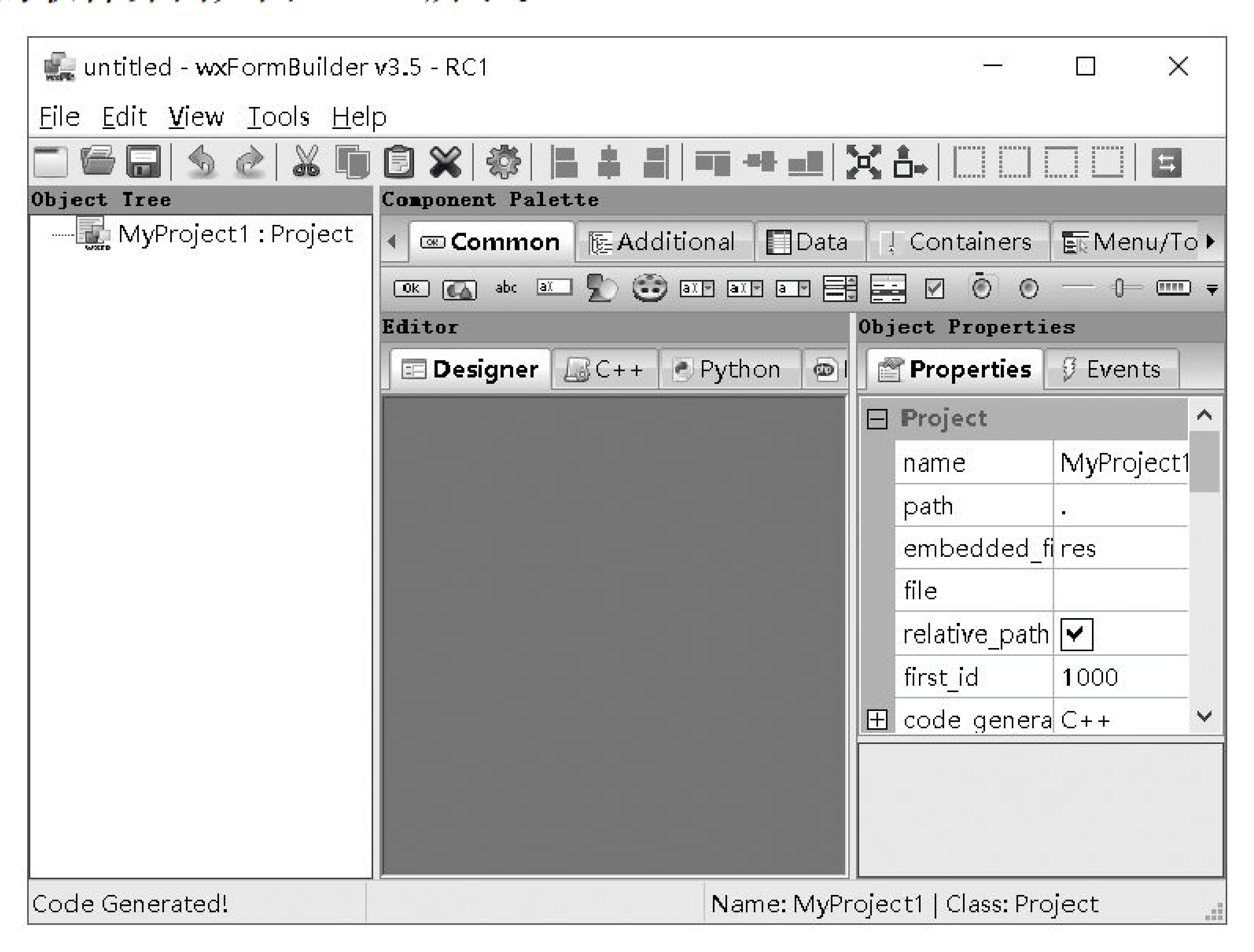


图 11-11 wxFormBuilder

在右侧的 ObjectProperties 子窗口中对项目做基本设置(见图 11-12),比如项目的名字(name)、保存的路径(path),记得选择 Python 作为代码生成语言(code_generation)。

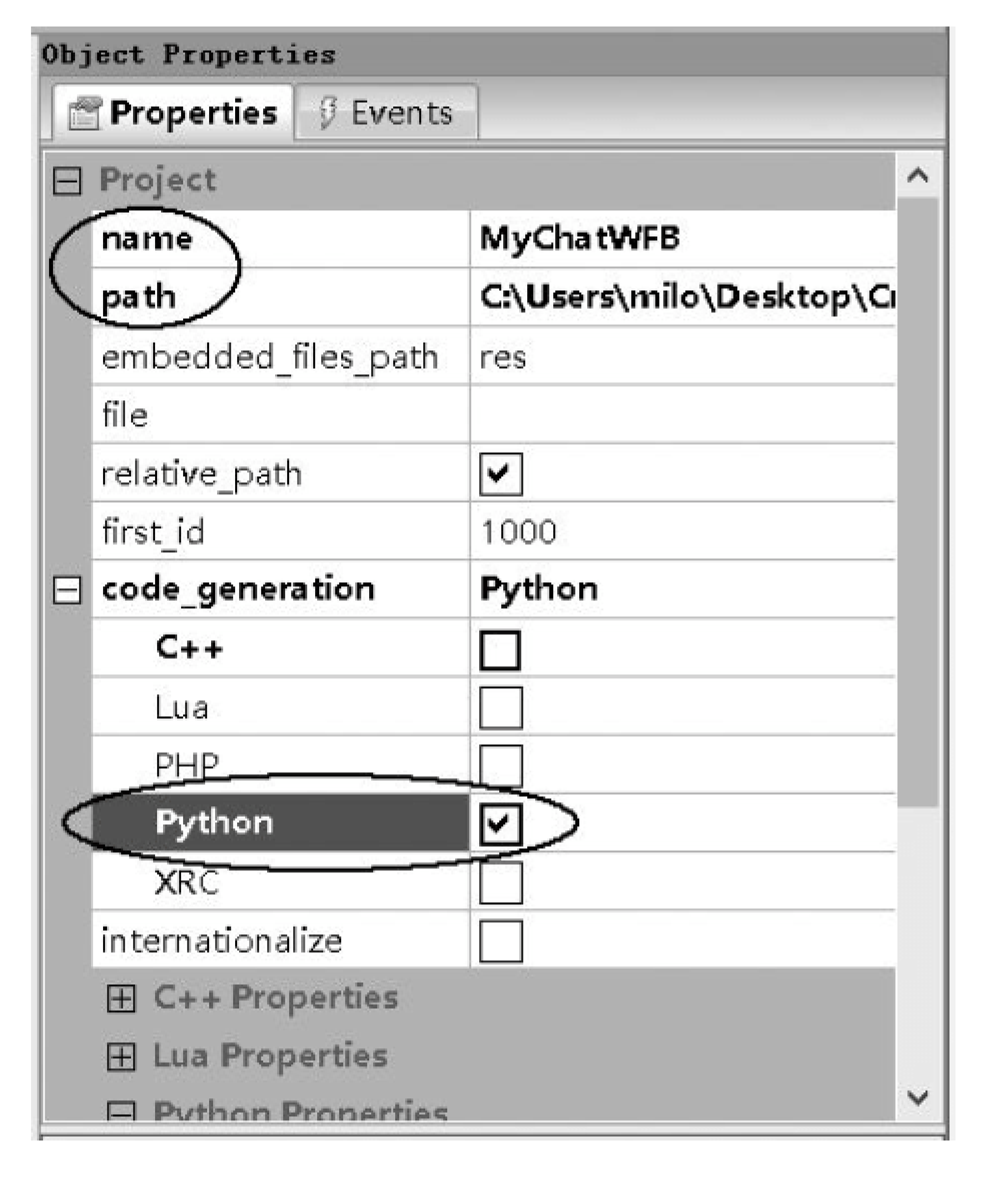


图 11-12　基本项目信息

以聊天窗口为例,对应地看一下如何用 wxFormBuilder 进行开发。注意图 11-13 中的箭头位置。

1. 生成空白窗口

依次单击 Forms(箭头 1)、Frame(箭头 2)按钮生成空白窗口 3,可以在右侧的 Object Properties 子窗口中对空白窗口做相应设置,比如 title(箭头 4)(见图 11-13)。

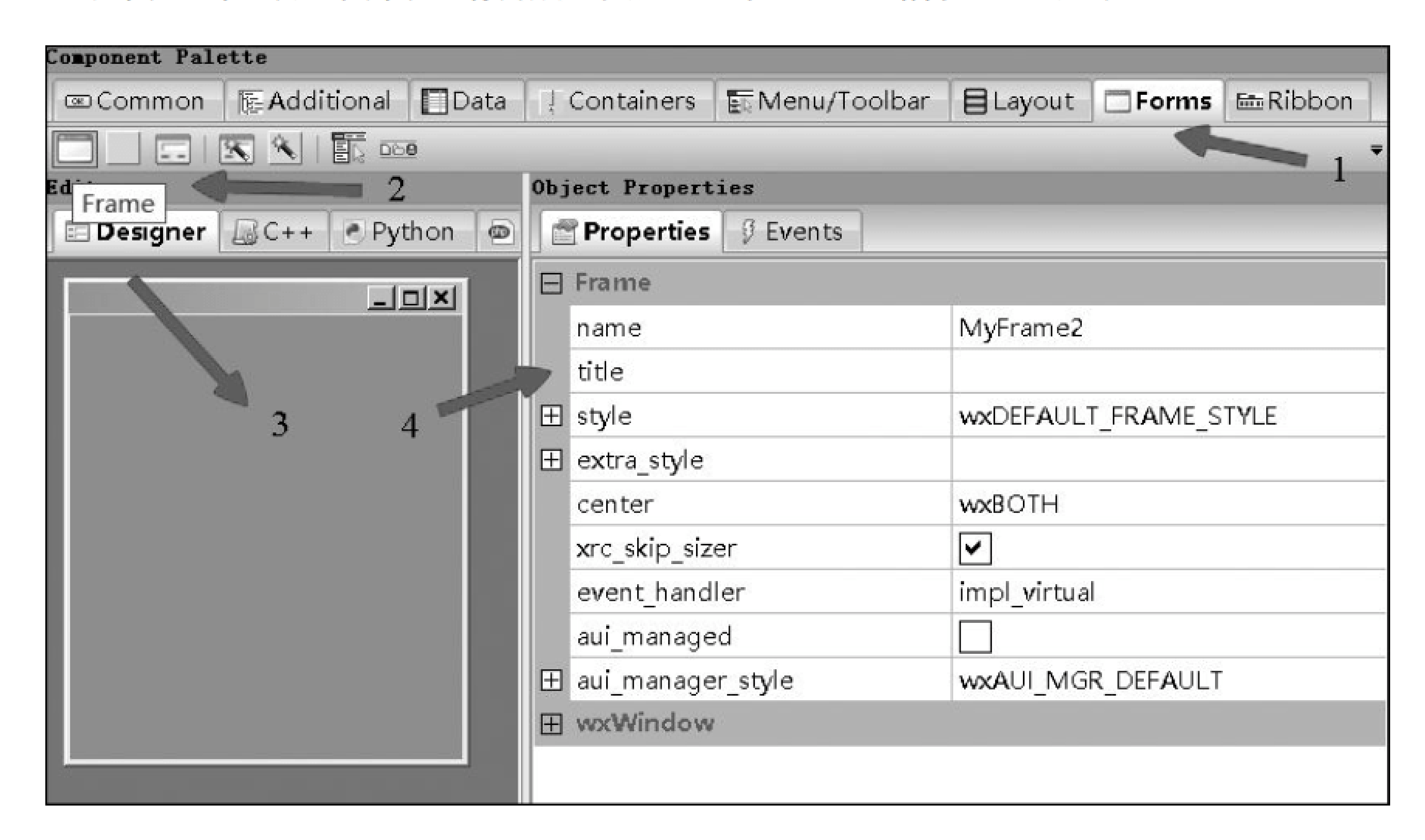


图 11-13　空白窗口

2. 布局管理

通过 Layout(箭头 1)添加 Sizer(箭头 2)进行布局管理,生成子目录(箭头 3),选择纵向(箭头 4)(见图 11-14)。

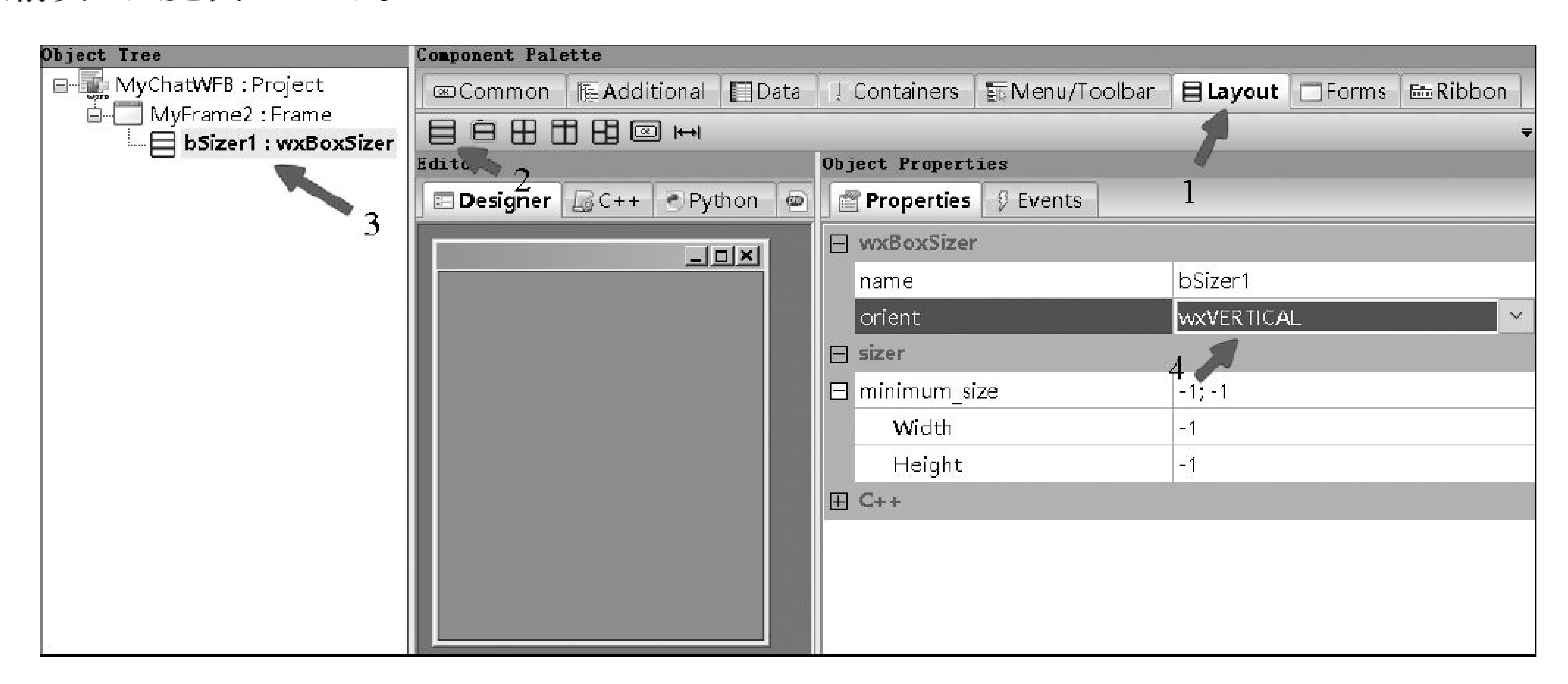


图 11-14 增加 Sizer

3. 添加组件

首先注意最后一行是两个横向按钮(见图 11-15),需要按照布局管理的操作(见图 11-14),先添加一个布局管理(箭头 1)并且在 Object Properties 子窗口中将 orient 选项设置为 wxHORIZONTAL(表示横向排列),然后选择该布局管理器(箭头 1),再在 Common 中找到所需要的组件依次单击添加,如在这个新的布局中添加 OK 按钮(箭头 2),如图 11-15 所示。

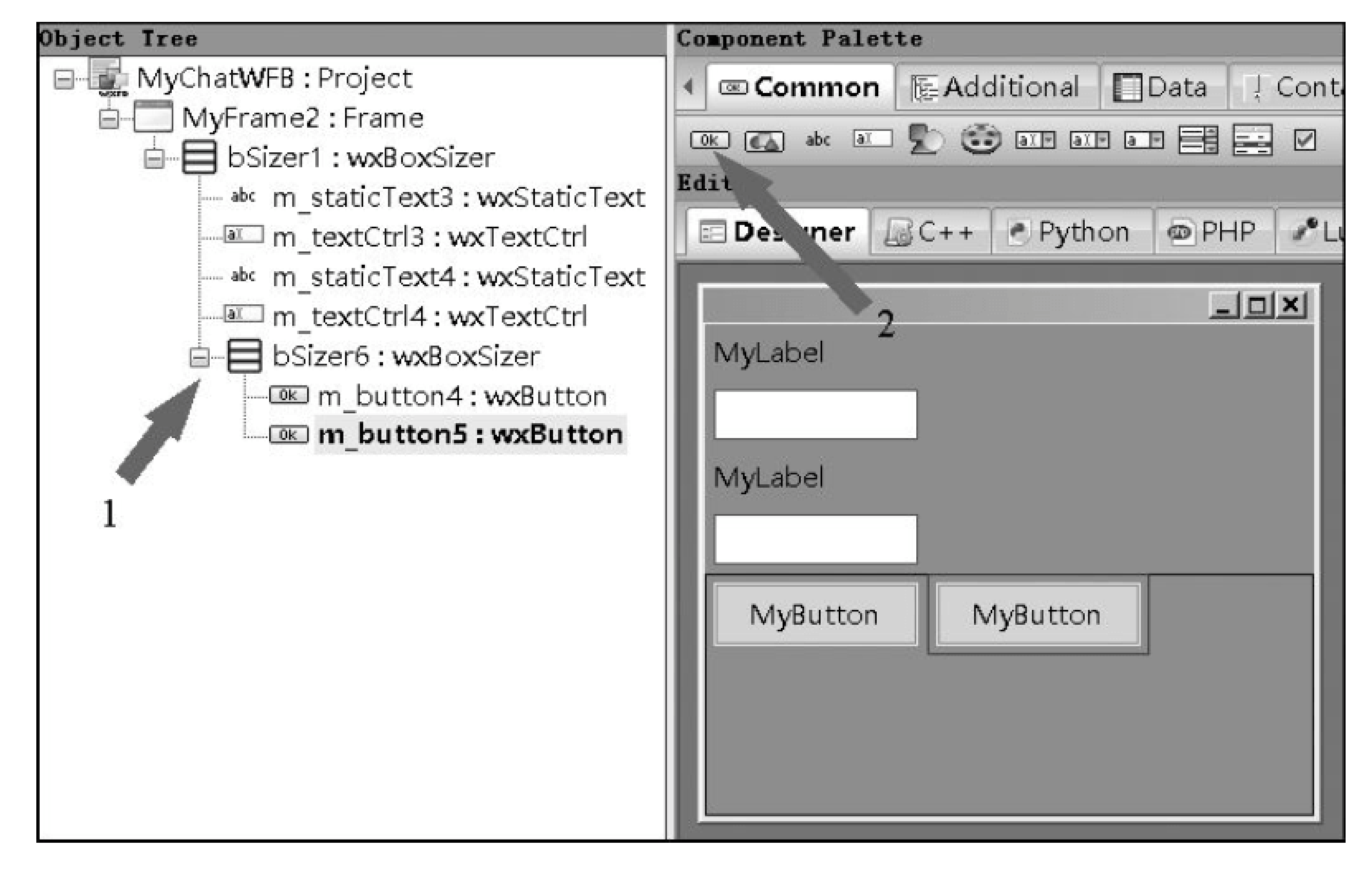


图 11-15 添加组件

4. 设置组件属性

所有窗口包括组件属性都在单击选择后，均可在其对应的 Object Properties 中找对应的属性进行设置(见图 11-16)，以聊天记录文本框为例，单击 allText：wxTextCtrl(箭头 1)目录可对文本框(箭头 2)进行设置，所有设置内容在右侧 Object Properties 窗口都会显示，如右侧窗口 4 个箭头所指的就是我们在编码阶段设置的一些属性。

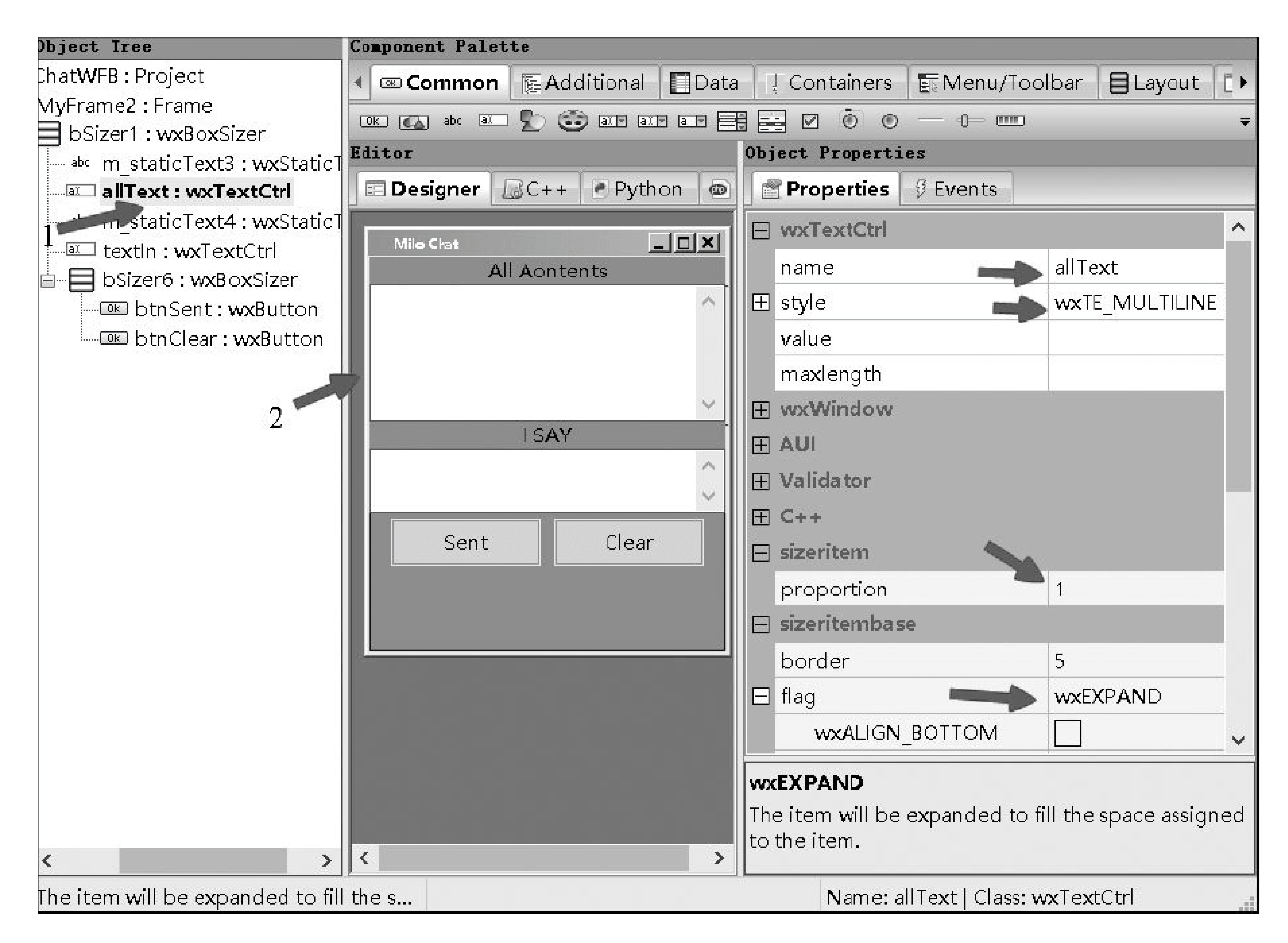


图 11-16 对象属性设置

5. 事件绑定

选择要触发事件的组件(箭头 1)，然后单击 Events 对其进行函数绑定(箭头 2)，不同的组件会有不同的事件触发机制，在这里按钮就是 OnButtonClick(箭头 3)，后面的文本框中输入你要定义的响应函数的名字，如图 11-17 所示。

到这里，工具能做的事情就介绍完了，接下来响应函数的逻辑代码需要你自己来写了。刚刚建立起来的框架，只要单击 Editor 子窗口中的 Python 标签就能看到代码了，但是不能在这里直接编辑，可以把代码复制后保存到一个 Python 脚本中。也可以直接将现在这个项目保存下来，按 F8 键可以直接生成代码文件。找到刚刚绑定事件时生成的响应函数，然后编写实现的方法即可。

F8 默认生成的文件名是 noname. py，位于开始时我们设置的项目路径中。打开后就是下面这个代码，没有重新排版。在此，我新导入了一个 time 模块，在 OnButtonSent 中编写了响应代码，如果你也动手编写了，在原代码块中需要注意缩进问题，默认生成的是 Tab 缩进，我们之前的习惯都是 4 个空格。最后编好逻辑代码直接运行就可以了，跟手写的代码是一样的效果。



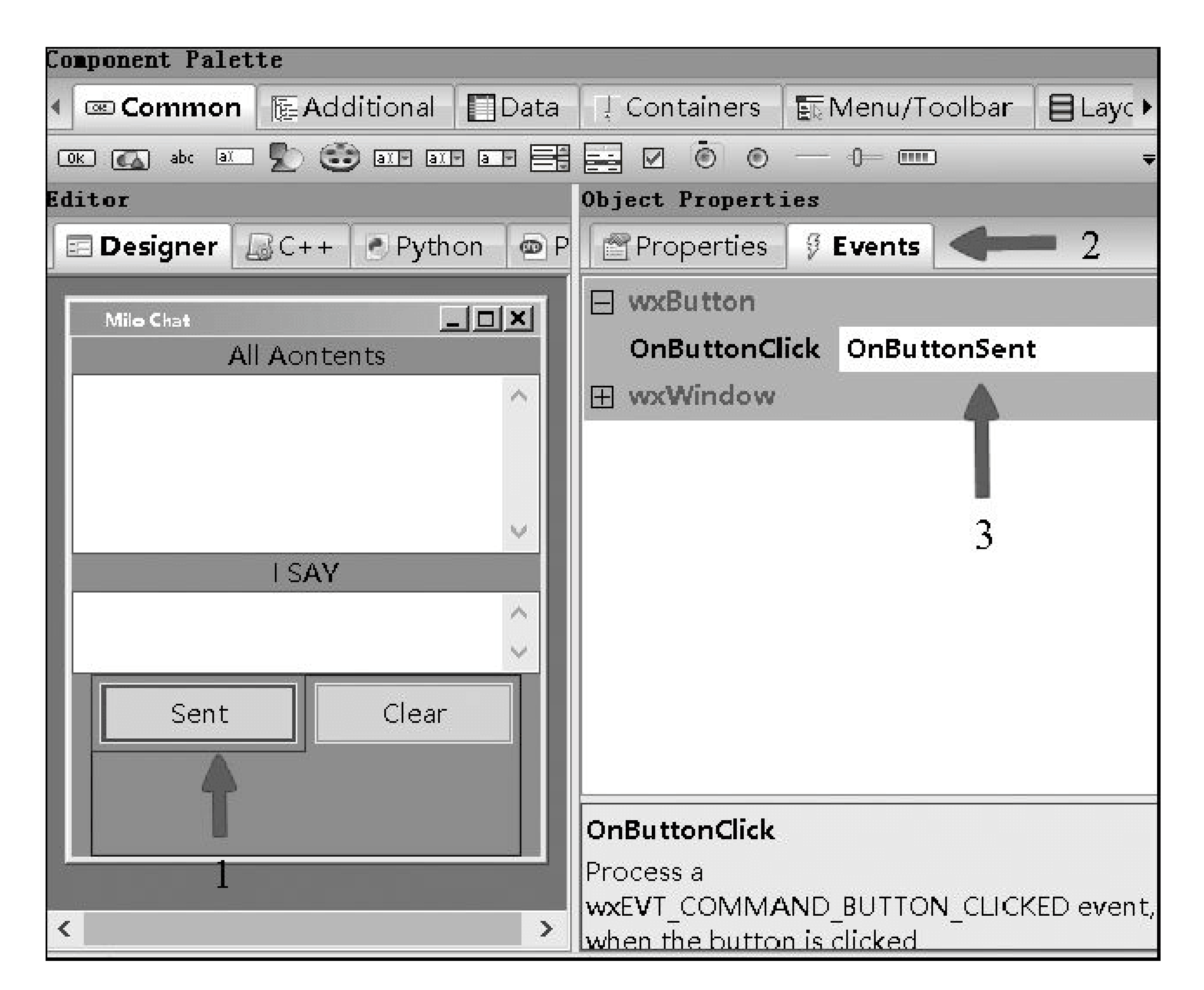


图 11-17　绑定按钮事件

```
#-*-coding: utf-8-*-

###########################################################################
## Python code generated with wxFormBuilder (version Jun 17 2015)
## http://www.wxformbuilder.org/
##
## PLEASE DO "NOT" EDIT THIS FILE!
###########################################################################

import wx
import wx.xrc
import time     #new

##################################################################
Class MyFrame2
##################################################################

class MyFrame2 (wx.Frame):

    def __init__(self, parent):
        wx.Frame.__init__ (self, parent, id = wx.ID_ANY, title = u"Milo Chat", pos =
wx.DefaultPosition, size = wx.Size(412,325), style = wx.DEFAULT_FRAME_STYLE|wx.TAB_
TRAVERSAL)

        self.SetSizeHintsSz(wx.DefaultSize, wx.DefaultSize)

        bSizer1 = wx.BoxSizer(wx.VERTICAL)

        self.m_staticText3 = wx.StaticText(self, wx.ID_ANY, u"All Aontents", wx.
DefaultPosition, wx.DefaultSize, wx.ALIGN_CENTRE)
```

```
        self.m_staticText3.Wrap(-1)
        bSizer1.Add(self.m_staticText3, 0, wx.EXPAND, 5)

        self.allText = wx.TextCtrl(self, wx. ID _ ANY, wx. EmptyString, wx.
DefaultPosition, wx.DefaultSize, wx.TE_MULTILINE)
        bSizer1.Add(self.allText, 1, wx.EXPAND, 5)

        self.m_staticText4 = wx.StaticText(self, wx. ID _ ANY, u" I SAY ", wx.
DefaultPosition, wx.DefaultSize, wx.ALIGN_CENTRE)
        self.m_staticText4.Wrap(-1)
        bSizer1.Add(self.m_staticText4, 0, wx.EXPAND, 5)

        self.textIn = wx.TextCtrl(self, wx. ID _ ANY, wx. EmptyString, wx.
DefaultPosition, wx.DefaultSize, wx.TE_MULTILINE)
        bSizer1.Add(self.textIn, 0, wx.EXPAND, 5)

        bSizer6 = wx.BoxSizer(wx.HORIZONTAL)

        self.btnSent = wx.Button(self, wx.ID_ANY, u"Sent", wx.DefaultPosition, wx.
DefaultSize, 0)
        bSizer6.Add(self.btnSent, 0, wx.ALL, 5)

        self.btnClear = wx.Button(self, wx.ID_ANY, u"Clear", wx.DefaultPosition,
wx.DefaultSize, 0)
        bSizer6.Add(self.btnClear, 0, wx.ALL, 5)

        bSizer1.Add(bSizer6, 1, wx.ALIGN_CENTER_HORIZONTAL, 5)

        self.SetSizer(bSizer1)
        self.Layout()

        self.Centre(wx.BOTH)

        #Connect Events
        self.btnSent.Bind(wx.EVT_BUTTON, self.OnButtonSent)
        self.btnClear.Bind(wx.EVT_BUTTON, self.OnButtonClear)

    def __del__(self):
        pass

    #Virtual event handlers, overide them in your derived class
    def OnButtonSent(self, event):
        event.Skip()
        """new"""
        userinput = self.textIn.GetValue()
        self.textIn.Clear()
        nowtime = time.ctime()
        inmsg = "You (%s) :\n%s \n" % (nowtime,userinput)
        self.allText.AppendText(inmsg)
```

```
    def OnButtonClear(self, event):
        event.Skip()

if __name__ == "__main__":
        app = wx.App()
        frame = MyFrame2(None)
        frame.Show()
        app.MainLoop()
```

11.5 生成可执行的二进制文件



生成可执行程序

现在，你已经编写了一个带图形化用户界面的软件。那么，接下来是不是就要分发给其他人用了呢？如果你已经迫不及待地用 U 盘 copy 到了其他人的计算机上，可能会让你失望，因为有可能其他人的计算机上并没有 Python 的运行环境，怎么办？给他安装 Python 环境然后再运行？

如果想在没有 Python 环境的计算机中运行 Python 脚本，可以将 Python 脚本打包封装成 Windows 可执行文件，就可以双击直接运行了。

用来打包生成可执行文件的工具很多，比如 py2exe、pyinstaller、cx_freeze，不一定哪个对你的版本支持得最好，所以有时候需要你多试试。这里我们用 pyinstaller。

安装 pyinstaller 的方法如下。

pyinstaller 的官方网站 http://www.pyinstaller.org 有详细的说明和手册，安装比较简单，可以直接用 pip install pyinstaller 安装升级（见图 11-18）。

```
Windows PowerShell
----> 1 import win32api

ModuleNotFoundError: No module named 'win32api'

In [13]: exit
PS C:\Users\milo> pip install pyinstaller
Collecting pyinstaller
  Downloading PyInstaller-3.3.tar.gz (3.5MB)
    100% |████████████████████████████████| 3.5MB 131kB/s
Requirement already satisfied: setuptools in c:\users\milo\appdata\local\programs\python\python36-32\lib\site-packages (from pyinstaller)
Collecting pefile>=2017.8.1 (from pyinstaller)
  Downloading pefile-2017.9.3.tar.gz (61kB)
    100% |████████████████████████████████| 71kB 227kB/s
Collecting macholib>=1.8 (from pyinstaller)
  Downloading macholib-1.8-py2.py3-none-any.whl
Collecting future (from pyinstaller)
  Downloading future-0.16.0.tar.gz (824kB)
    100% |████████████████████████████████| 829kB 217kB/s
Collecting pypiwin32 (from pyinstaller)
  Downloading pypiwin32-220-cp36-none-win32.whl (8.3MB)
    100% |████████████████████████████████| 8.3MB 67kB/s
Collecting altgraph>=0.13 (from macholib>=1.8->pyinstaller)
  Downloading altgraph-0.14-py2.py3-none-any.whl
Installing collected packages: future, pefile, altgraph, macholib, pypiwin32, pyinstaller
  Running setup.py install for future ... done
  Running setup.py install for pefile ... done
  Running setup.py install for pyinstaller ... done
Successfully installed altgraph-0.14 future-0.16.0 macholib-1.8 pefile-2017.9.3 pyinstaller-3.3 pypiwin32-220
PS C:\Users\milo>
```

图 11-18　安装 pyinstaller

安装后切换到脚本所在目录（cd），执行 pyinstaller -F noname.py 命令就可以生成可执行文件了（见图 11-19），F 选项的作用是生成一个可执行文件，如果不加，会生成一堆零散

```
命令提示符
17242 INFO: Building PYZ (ZlibArchive) C:\Users\milo\Desktop\CrazyPythonZeroToH
ero\code\11\build\noname\out00-PYZ.pyz completed successfully.
17263 INFO: checking PKG
17264 INFO: Building PKG because out00-PKG.toc is non existent
17265 INFO: Building PKG (CArchive) out00-PKG.pkg
23685 INFO: Building PKG (CArchive) out00-PKG.pkg completed successfully.
23691 INFO: Bootloader c:\users\milo\appdata\local\programs\python\python36-32\
lib\site-packages\PyInstaller\bootloader\Windows-32bit\run.exe
23691 INFO: checking EXE
23691 INFO: Building EXE because out00-EXE.toc is non existent
23692 INFO: Building EXE from out00-EXE.toc
23692 INFO: Appending archive to EXE C:\Users\milo\Desktop\CrazyPythonZeroToHer
o\code\11\dist\noname.exe
23712 INFO: Building EXE from out00-EXE.toc completed successfully.

C:\Users\milo\Desktop\CrazyPythonZeroToHero\code\11>pyinstaller -F noname.py
```

图 11-19　生成一个可执行文件

文件。

成功后会生成一个 dist 目录，里面的 noname. exe 就是可以双击运行的可执行文件了（见图 11-20）。

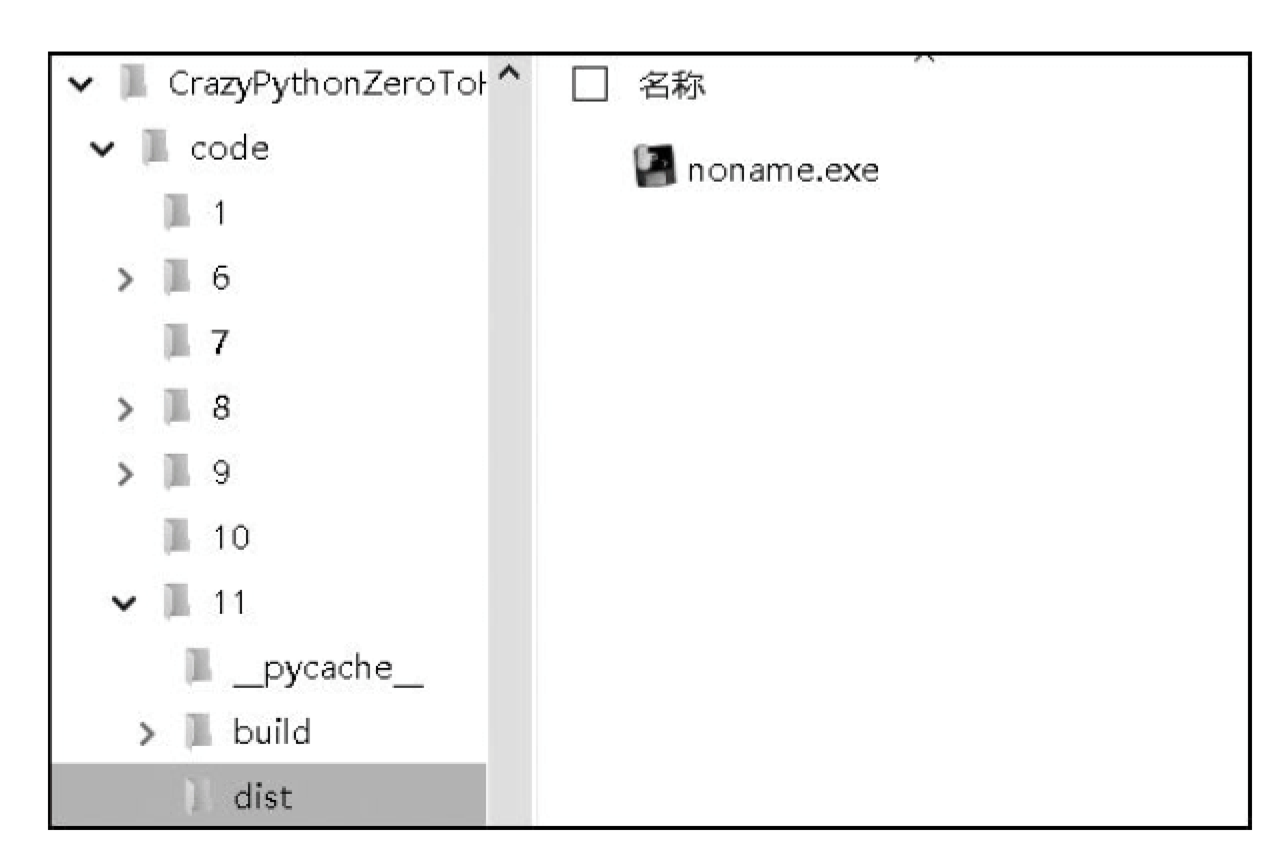


图 11-20　可执行文件

如果你有. ico 类型的图标文件，还可以拿来生成个性图标的可执行文件。怎么做？请自己动手查手册。

重点提示

在这一章中，你要掌握的内容：

（1）理解 GUI 编程中的窗口、事件、组件等概念。

（2）掌握 wxPython 的子类化开发模式。

动动手

（1）实现一个简易的记事本，带有打开和保存功能。

（2）做一个简易计算器，布局要求除了 0～9 十个数字按键还要包含小数点以及加、减、乘、除和等号，一共 16 个按键，合理设计布局。最后生成可执行的. exe 程序。



第 12 章 Python 玩转数据库

开发应用软件或网站时，大部分情况都会用到数据库存储和管理数据。数据库并不是 Python 自身特有的，数据库在计算机领域是一个独立的存在，而且有很多公司、机构开发的不同版本。

12.1 数据库初始

数据库按数据存储方式区分，目前用到最多的就是关系型数据库和非关系型数据库。

关系型数据库用一句话概括，就是数据存储的方式像列表一样，按一定的结构关系存取数据。主流的关系型数据库如 SQLite、MySQL、Oracle、Access。虽然数据库的版本不同，但针对数据库的操作主要就是数据的增、删、改、查，而且有一套通用的、成熟的语句可以用在不同版本的数据库上，叫作 SQL，即结构化查询语言（Structured Query Language）。

Python 对关系型数据库都提供了标准数据库接口 Python DB-API。Python DB-API 为开发人员提供了数据库应用编程接口，不同的数据库需要下载不同的 DB-API 模块，例如你需要访问 Oracle 数据库和 MySQL 数据，就需要下载 Oracle 和 MySQL 数据库模块。可以在 https://wiki.python.org/moin/DatabaseInterfaces 查看详细的支持数据库列表以及相应的模块。

DB-API 是一个规范，它定义了一系列必需的对象和数据库存取方式，以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口。使用它连接各数据库后，就可以用相同的方式操作各数据库。Python DB-API 使用流程如下。

（1）引入 DB-API 模块。

（2）获取与数据库的连接。

（3）执行 SQL 语句和存储过程。

（4）关闭数据库连接。

相对于关系型数据库的就是非关系型数据库 NoSQL。NoSQL 数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题。

因为过多的数据库探讨已经超过本书的范围，比如 MySQL 就够写一本书了，所以这里通过两个有代表性的关系型数据库，来学习一下 Python 如何使用数据库。

12.2 SQLite 数据库

SQLite 是一款非常流行的关系型数据库，由于它非常轻盈，因此被大量应用程序广泛使用。最主要的是 SQLite 内嵌在 Python 中，不必额外安装。要使用 SQLite 只需要导入 Python 标准发行版中自带的模块 sqlite3 就可以了。SQLite 数据库既可以直接保存到文件中，也可以保存到内存中。


玩转数据库 sqlite3

要开发基于数据库的程序，需要先掌握数据库的一些基本知识。SQLite 属于关系型数据库，你可以将数据库联想成存储数据的表格，数据库的基本操作包括创建一个数据库、创建表、增加（插入）记录、删除记录、修改记录以及查询记录。数据库的这些操作都是通过 SQL 语句实现的，下面的例子中主要 SQL 语句如下。

（1）创建数据库

```
create database 库名
```

（2）创建表

```
create table 表名(字段名字段选项[,字段名字段选项...])
```

（3）删除表

```
drop table 表名
```

（4）插入记录

```
insert into 表名[字段名列表] values(值列表)
```

（5）删除记录

```
delete from 表名 where 删除条件
```

（6）修改记录

```
update 表名 set 字段名 =值[, 字段名 =值...] where 修改条件
```

（7）查询表所有记录

```
Select * from 表名
```

这些都是在数据库中使用的标准 SQL 语句，下面直接通过 sqlite3 看一下 Python 如何实现这些操作。

sqlite3 基本操作如下所述。

首先需要导入模块并创建连接对象，创建连接对象有点像文件操作，这时只是建立了与数据库的一个通道，返回的是一个数据库连接对象。数据库文件如果存在，则直接打开；

不存在，则直接创建。

```
>>>import sqlite3
>>>conn = sqlite3.connect('hero.db')
>>>import os
>>>os.getcwd()
'C:\\Users\\milo\\AppData\\Local\\Programs\\Python\\Python36-32'
>>>os.listdir()
['hero.db']
```

连接对象的几个主要方法如下。

execute(SQL 语句)：执行 SQL 语句。

cursor()：创建一个游标。

commit()：事务提交。

rollback()：事务回滚。

close()：关闭数据库连接。

创建了数据库连接后，接下来需要通过连接对象的 cursor 方法创建一个游标对象。游标对象对表中数据的检索提供了灵活的方法，可以通过游标对象的 execute 方法执行 SQL 语句，实现增、删、改的操作，通过 fetchall()和 fetchon()返回查询结果。

在刚才创建的 hero 库中创建表格 player，并插入 4 条数据。记得提交事务，否则数据库不会同步。

```
>>>cur = conn.cursor()
>>>cur.execute("create table player (name VARCHAR(30), age INT,sex CHAR(1))")
                    #创建表
<sqlite3.Cursor object at 0x05974960>
>>>cur.execute("insert into player VALUES('milo', 18, 'M')")
<sqlite3.Cursor object at 0x05974960>
>>>cur.execute("insert into player VALUES('zou', 20, 'M')")
<sqlite3.Cursor object at 0x05974960>
>>>cur.execute("insert into player VALUES('qi', 21, 'F')")
<sqlite3.Cursor object at 0x05974960>
>>>cur.execute("insert into player VALUES('xian', 21, 'F')")
<sqlite3.Cursor object at 0x05974960>
>>>conn.commit()      #提交事务相当于将 SQL 语句传给数据库执行
```

这里创建表的 SQL 语句的意思是创建一个名为 player 的表，有 name、age、sex 三个字段，每个字段的保存类型分别是 VARCHAR(30)(最长 30 的可变长度字符串)、INT(整数)和 CHAR(1)(固定以为字符)。

```
create table player(name VARCHAR(30), age INT,sex CHAR(1))
```

有了数据就可以进行修改、删除和查询操作了。我们修改 milo 的年龄为 19，删除 zou 的记录，分两种方式查询数据库数据。

```
>>>cur.execute("update player set age=19 where name='milo'")
```

```
<sqlite3.Cursor object at 0x05974960>
>>>cur.execute("delete from player where name = 'zou'")
<sqlite3.Cursor object at 0x05974960>
>>>cur.execute("select * from player")      #执行查询命令
<sqlite3.Cursor object at 0x05974960>
>>>one = cur.fetchone()                      #获得一条数据
>>>print(one)
('milo', 19, 'M')
>>>all = cur.fetchall()                      #获得多条数据
>>>print(all)
[('qi', 21, 'F'), ('xian', 21, 'F')]
```

需要注意的是，执行过查询命令后返回的不是数据库的数据，而只是一个游标对象，需要在调用游标对象的 fetchone 和 fetchall 来获取数据。

最后需要关闭游标、关闭数据库连接，这就像文件连接一样，数据库用过之后也需要关闭，先关游标再关数据库。

```
>>>cur.close()
>>>conn.close()
```

SQLite 基本可以满足现阶段一般的开发需求了，对于其他数据库，用 Python 模块操作也非常方便，只不过，数据库的学习成本会比较高。接下来我们直接用 SQLite 的例子来看一下在其他数据库上的操作过程。

12.3 Python 连接 MySQL

MySQL 是最流行的关系型数据库管理系统，在 Web 应用方面 MySQL 是最好的关系数据库管理系统应用软件之一。



玩转数据库
MySQL-Python

与 SQLite 不同，MySQL 是以 C/S(客户端/服务端)的结构来实现的，需要安装服务端和客户端软件。服务端有一个守护进程保证 MySQL 处于运行状态，客户端则通过指定主机、用户名和密码的方式连接到服务端进行操作。客户端可以通过终端访问，也可以通过图形工具管理。

我们不对 MySQL 做过多介绍，直接来看如何在 Python 中操作。首先确保 MySQL 守护进程已经启动，然后安装模块开始操作，Python 3 中安装 PyMySQL，Python 2 中安装 MySQLdb(具体使用是一样的，只是模块名字不一样)。

通过 pip 安装 PyMySQL(见图 12-1)。

安装完成后可以在 PythonShell 中导入模块试一下，没有报错就是成功了。

```
>>>import pymysql
```

在确保 MySQL 已经正常启动且可用的状态下，我们来看一下跟 SQLite 例子相同的操作在 MySQL 中怎样实现。





```
或批处理文件。

C:\Users\milo\Desktop\CrazyPythonZeroToHero\code\11>pip install PyMySQL
Collecting PyMySQL
  Downloading https://files.pythonhosted.org/packages/e5/07/c0f249aa0b7b0517b58
43eeab689b9ccc6a6bb0536fc9d95e65901e6f2ac/PyMySQL-0.8.0-py2.py3-none-any.whl (8
3kB)
    36% |███████████▌                    | 30kB 207kB/s eta 0:00:01
    49% |███████████████▊                | 40kB 181kB/s eta 0:0
    61% |███████████████████▋            | 51kB 218kB/s eta
    73% |███████████████████████▌        | 61kB 257kB/s
    86% |███████████████████████████▍    | 71kB 285
    98% |███████████████████████████████▍| 81kB
    100% |████████████████████████████████| 92k
B 315kB/s
```

图 12-1　安装 PyMySQL

```
#pymysql_eg.py
import pymysql
conn = pymysql.connect(host = "localhost",#MySQL 服务端地址
                       user = "root",        #登录 MySQl 的用户名
                       password = "123456", #密码
                       db = "hero")          #要连接的库名

cur = conn.cursor()                          #创建游标
sqlCT = "create table player (name VARCHAR(30), age INT,sex CHAR(1))"
cur.execute(sqlCT)                           #创建表

cur.execute("insert into player VALUES ('milo', 18, 'M')")
cur.execute("insert into player VALUES ('zou', 20, 'M')")
cur.execute("insert into player VALUES ('qi', 21, 'F')")
cur.execute("insert into player VALUES ('xian', 21, 'F')")
conn.commit()                                #事务提交

cur.execute("update player set age= 19 where name = 'milo'")  #修改数据
cur.execute("delete from player where name = 'zou'")  #删除数据
cur.execute("select * from player")         #执行查询命令

one = cur.fetchone()                         #获得一条数据
all = cur.fetchall()                         #获得多条数据
print(one)
print(all)

cur.close()                                  #关闭游标
conn.close()                                 #关闭数据库连接
```

有看出异同吗？除了模块不同，名字不同，以及连接数据库时指定了主机名、用户名、密码等之外，所有的操作都是一样的，这也是标准数据库接口(Python DB-API)的好处，换了数据库只需要更改模块或者做少量更改就可以了。

针对数据库的操作还有很多便捷的方法，掌握了基本原理后，在工作中摸索出适合你生产环境的工具就好了。

重点提示

在这一章中,你要掌握的内容:

(1) 掌握基本的 SQL 语句,会进行基本的增删改查操作。

(2) 熟悉 Python 操作数据库的方式。

动动手

(1) 执行第 12.2 节 SQLite 的所有操作。

(2) 编写 Console 下的信息管理系统(可以是各种信息,比如图书、商品等)。建议及要求如下。

① 可以写一个员工信息管理系统。

② 程序需要具备数据增、删、改、查功能,增、删、改、查四项功能要写成四个函数。

③ 有四个字段,员工的名字、性别、年龄、电话,存储形式采用 sqlite3。

④ 存储结构自行设计。

⑤ 不要赶工期,代码要做到简洁、优美、规范、注释到位。

⑥ 也可自行设计类似的软件,功能不少于以上所涉及的内容。

(3) 给题(2)中的信息管理系统做一个 GUI。

第 13 章 分身有术：多线程编程

随着计算机硬件的发展，计算能力越来越强，程序运行也越来越快。现代操作系统都是支持“多任务”的操作系统，不只是计算机，手机现在也在向多核发展。最新的手机 CPU 已经达到了 8 核，8 核描述的是手机 CPU 的处理核心，可以同时有 8 个线程进行运算。多任务的情况下，多核的优势就很明显了，比如手机上运行大型游戏、视频解码等。

在进行多线程学习之前，可以想象一些场景，比如前面写的《英雄无敌》，现在只是个回合制的游戏，即在地图上或者开战时，英雄和敌人是一人攻击一回合。但如果想让英雄和敌人都能同时在地图上移动，各自独立开来，比如玩家控制英雄移动，敌人自行在地图上随机或按一定规律移动，或者同时有几百个人在地图上移动，这时总不能让一个人动完之后再让另一个人动。这种情况就需要多线程来支持了。

13.1 进程与线程

要想学习多线程编程，首先要弄清楚进程和线程的一些相关概念。



进程与线程

(1) 进程：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。狭义来说，进程就是一个程序执行时的实例，动态的执行过程存在于计算机的内存中，占用 CPU 资源。如果你在 Windows 系统中用过进程管理，那么你对这个词可能并不陌生。

(2) 程序：要说进程，就不得不说程序。到目前为止，我们都在学习怎么写程序，其本身没有任何运行的含义，是一个静态的概念。而进程则是在处理机上的一次执行过程，它是一个动态的概念。这个不难理解，其实进程是包含程序的，进程的执行离不开程序，进程中的文本区域就是代码区，也就是程序。

(3) 线程：通常在一个进程中可以包含若干个线程，一个程序至少有一个线程，若程序只有一个线程，那就是程序本身。线程可以利用进程所拥有的资源，在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位，由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效地提高系统多个程序间并发执行的程度。

(4) 多线程：在一个程序中，这些独立运行的程序片段叫作线程(Thread)，利用线程编

程的概念就叫作多线程处理。多线程是为了同步完成多项任务,不是为了提高运行效率,而是为了提高资源使用效率来提高系统效率。线程是在同一时间需要完成多项任务时实现的。

多线程的出现就是为了提高效率。你可以想象一下,火车至少会有一节车厢(车头),运行时多节车厢同时启动,多线程的每个线程就像火车的每一节车厢,而进程则是火车,想要多运货就增加车厢。

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间,一个进程崩溃后,在保护模式下不会对其他进程产生影响,而线程只是一个进程中的分支任务。线程有自己的堆栈和局部变量,但线程之间没有单独的地址空间,一个线程死掉就等于整个进程死掉,所以多进程的程序要比多线程的程序健壮,但在进程切换时,耗费资源较大,效率要差一些。

对于一些要求同时进行并且又要共享某些变量的并发操作,只能用线程,不能用进程。具体情况如下。

(1) 一个程序至少有一个进程,一个进程至少有一个线程。

(2) 线程的划分尺度小于进程,使得多线程程序的并发性高。

(3) 进程在执行过程中拥有独立的内存单元,多个线程共享内存,极大地提高了程序的运行效率。

(4) 线程在执行过程中与进程的区别在于,每个独立的线程有一个程序运行的入口,顺序执行序列和程序的出口。但是线程不能够独立执行,必须依存在应用程序中,由应用程序提供多个线程执行控制。

(5) 从逻辑角度来看,多线程的意义在于一个应用程序中有多个执行部分可以同时执行。但操作系统并没有将多个线程看作多个独立的应用,来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

线程和进程在使用上各有优缺点:线程执行开销小,但不利于资源的管理和保护;而进程正相反。同时,线程适合在多核处理机机器上运行,而进程则可以跨机器迁移。

13.2 多线程

Python 3 中提供了一个 threading 模块用来支持多线程编程,在 Python 2 中除了 threading 还有一个多线程模块 thread。其实 threading 是对 thread 的封装,一般只用 threading 就可以了。



多线程 threading

13.2.1 创建线程

创建线程之前,先假设一个场景吧,比如播放器放视频时是影像和声音同时输出的,那么在这个程序的进程中至少有处理声音和影的两个线程。我们用文字输出分别代表影像和声音。

对于多线程编程,我们可以直接利用 threading. Thread 类直接生成线程对象来生成线程。



```
#th_1.py
import threading        #导入 threading 模块
def run(x,y):
    for i in range(y):
        print("输出 %s 共 %d 次"%(x,y))

#生成两个线程图 t1 和 t2
t1 = threading.Thread(target = run, args = ("画面", 3))
t2 = threading.Thread(target = run, args = ("声音", 3))

#分别启动两个线程
t1.start()

t2.start()
"""
run("画面",3)
run("声音",3)
"""
```

这里我们定义的函数 run，如果像注释中那样直接被调用两次，则会依次执行，然后输出如下：

```
输出 画面 共 3 次
输出 画面 共 3 次
输出 画面 共 3 次
输出 声音 共 3 次
输出 声音 共 3 次
输出 声音 共 3 次
```

这就是一个普通的按顺序执行的过程，并没有产生多线程，而如果我们生成 t1、t2 两个线程来执行，效果就会类似下面这样：

```
输出 画面 共 3 次输出 声音 共 3 次
输出 画面 共 3 次输出 声音 共 3 次
输出 画面 共 3 次输出 声音 共 3 次
```

生成线程对象时的两个参数：target 是要调用的函数，args 是给函数传递参数。除了直接生成对象，我们还要看一下怎样通过继承 threading 模块的 Thread 类定义一个新的类。在新类中重载 run 方法，再通过 start 方法创建线程。run 方法会在线程创建后自动运行。

```
#th_2.py
import threading
class MyThread(threading.Thread):
    def __init__(self,x,y):                 #子类定义初始化方法
        threading.Thread.__init__(self)     #调用父类初始化方法
        self.x, self.y = x, y
    def run(self):                          #重载 run 方法
```

```
        for i in range(self.y):
            print("输出 %s 共 %d 次"%(self.x, self.y))

t1 = MyThread("画面", 3)
t2 = MyThread("声音", 3)

t1.start()
t2.start()
```

13.2.2 线程对象的方法

只是生成线程还不够,很多时候我们还要控制线程,这时就要依靠 Thread 对象的其他方法了。

1. isAlive

当我们创建了线程之后,有时需要对线程进行跟踪,可以通过 isAlive 方法判断线程是否运行,如果正在运行,返回 True。下面我们来生成一个可以持续 30 秒的线程,然后来查看它的状态,线程启动后,30 秒内查看,都是 True。

```
>>>import time
>>>import threading
>>>def run():
    print("Start")
    time.sleep(30)
    print("End")
>>>t = threading.Thread(target = run)
>>>t.start()
Start        #线程启动
>>>print(t.isAlive())
True
End          #线程已结束
>>>print(t.isAlive())
False
```

2. join

虽然线程的意义是为了同时启动多个任务,但有时在程序运行过程中启动了某一个线程后,必须等这个线程结束才能继续其他线程,此时调用 join 方法。

比如,创建线程的声音和画面的例子,在 t1.start()之后先执行 t1.join()再启动 t2,这样 t2 就要等到 t1 结束才会开始执行。

```
#th_join.py
import threading          #导入 threading 模块
def run(x,y):
    for i in range(y):
        print("输出 %s 共 %d 次"%(x,y))
```



```
#生成两个线程图 t1 和 t2
t1 = threading.Thread(target = run, args = ("画面", 3))
t2 = threading.Thread(target = run, args = ("声音", 3))

#分别启动两个线程
t1.start()
t1.join()
t2.start()
```

3. 线程名

我们可以给线程起一些好记易懂的名字加以区分，这样更易于控制，可以在构造函数中通过给 name 属性赋值来设置，也可以在创建线程对象后通过 setNmae 和 getName 方法来设置和获取。

```
#th_name.py
import threading
class MyThread(threading.Thread):
    def __init__(self,threadName):
        threading.Thread.__init__(self, name = threadName)

t1 = MyThread("画面")
print(t1.getName())
t1.setName('声音')
print(t1.getName())
```

4. setDeamon

启动线程后，主线程要退出时，会先检查子线程是否已完成，如果子线程未完成，则会等待子线程完成再退出。实际上很多软件都是关闭之后就直接终止一切线程，这种情况设置 daemon 属性为 True 就可以了。比如声音画面的线程本身可能要执行两个小时，但只要对线程对象的 daemon 属性值为 True，则进程结束时不会等待线程而直接退出。

可以改写声音和画面的小例子，让声音(t2)执行 60 秒，画面(t1)执行 20 秒，但是通过 setDaemon 设置 t2 的 daemon 属性为 True，动手试试看效果吧，t2 是否执行够 60 秒呢？

13.2.3　线程锁

多线程的程序有可能会涉及多个线程同时操作一个数据，如果不进行控制，就会哪个线程最后修改则保存哪个线程的修改。为保证数据的安全，这时就要加线程锁，也就是实现数据同步。


线程锁

最简单线程同步就是线程锁，其实是 threading 模块，Lock 类和 RLock 类都可以用来实现简单的线程同步。Lock 类和 RLock 类都提供了 acquire 方法和 release 方法多线程保护数据，简单说就是同一时间只能一个线程对数据进行操作。这样的线程只要放在 acquire 方法和 release 方法就可

以了。

下面我们模拟 5 个线程同时修改一个数据的情形来看一下如何通过 Lock 进行线程同步。

```
#th_syn.py
import threading
import time
class MyThread(threading.Thread):
    def run(self):
        global x_db
        lock.acquire()      #调用所对象方法加锁
        x_db += 1
        time.sleep(5)
        print(x_db)
        lock.release()      #调用锁对象解锁

lock = threading.Lock()     #生成 Lock 对象(生成锁)

x_db = 0
t5 = []

for i in range(5):
    tn = MyThread()
    t5.append(tn)

for j in t5:
    j.start()
```

因为有锁的控制,所以线程间并不会相互影响,每次只有一个线程在对数据进行修改,线程生存 5 秒,解锁之前看到的就是当前线程的修改结果,所有线程结束是 25 秒。效果如下:

```
1
2
3
4
5
```

但是,如果把代码中有注释的三行代码删除,则相当于没有线程同步控制,所有线程同时对数据进行操作,因为 5 秒的时间足够所有线程执行完毕了,所以当所有线程都执行后,数据已经累加到最大,此时第一个线程在 5 秒后输出的也是最后的结果,整个进程执行 5 秒左右,结果如下,每个线程的输出都是 5。

```
5
5
5
5
5
```

13.2.4 多线程的本质

学会了多线程的实现方法并不代表什么情况都要用，Python 多线程实际上有时并不会发挥它的作用，因为在 Python 通过全局解释锁来确保同一时间只能有一个线程执行任务，也就是说即便我们看到的执行效果好像是多个任务并发执行，但实际上在 CPU 上只是在交替执行而已，只不过 CPU 的运算速度足够快，感觉不出来罢了。也就是说如果是 8 核 CPU，那么多线程并不能充分利用上。所以，如果想要高效地利用多核 CPU 进行计算密集型任务，需要使用多进程。

虽然看起来有些鸡肋(因为在高并发计算中并没有优势，Python 不是什么都强大的)，但是如果是 I/O 密集型的工作，多线程就非常适合。因为 I/O 任务的重点在 I/O 上，并不在计算上，当一个任务阻塞在 I/O 操作上时，我们可以立即启动其他线程执行其他操作。

13.3 实例：批量主机扫描

设想一个从事运维工作的工作人员，管理着 100 台服务器。那么，在日常工作中如何快速判断这 100 台服务器的网络是否畅通呢？最简单的办法就是写个脚本通过 ping 命令来判断主机状态，但是 100 台一个一个的测试显然效率不高，所以，我们干脆直接通过多线程，相当于同时对 100 台主机 ping。下面就来看下这个程序。


多线程主机扫描

```
#!/usr/bin/python
#th_ping.py
from threading import Thread
import subprocess
from queue import Queue

queue5 = Queue()
ips = ['10.0.2.11',
       '10.0.2.12',
       '10.0.2.13',
       '10.0.2.14',
       '10.0.2.15']

def pinger(i, q):
    while True:
        ip = q.get()        #从队列中取出一个元素
        print("Thread %s : Pinging: %s " % (i, ip))
        """linux
        ret = subprocess.call('ping -c 1 %s' %ip,
                              shell= True,
                              stdout= open('/dev/null', 'w'),
                              stderr= subprocess.STDOUT)
        """
        #windows
```

```
        ret = subprocess.call('ping -c 1 %s' %ip,
                              shell = True,
                              stdout = open('NUL', 'w'),
                              stderr = subprocess.STDOUT)
        if ret == 0:
            print('%s : is alive' %ip)
        else:
            print('%s : did not respond' %ip)
        q.task_done()
        #占位,在完成一项任务之后,向任务已经完成的队列发送一个信号

for i in range(3):
    th = Thread(target = pinger, args = (i, queue5))
    th.setDaemon(True)         #设置 daemon 属性为 True
    th.start()

for ip in ips:
    queue5.put(ip)             #向对列添加元素

print("Main Thread waiting...")
queue5.join()                  #阻塞调用线程,直到队列中的所有任务被处理掉
print('Done')
```

在这段代码中,我们将主机数设置为 5 个,实际 100 个的原理是一样的,你也可以将主机 ip 存储到数据库中进行管理。创建了 3 个线程来处理这 5 个 ping,你也可以创建 5 个。

这里用到两个新的模块:subprocess 和 queue。

subprocess 的作用是用来管理子进程,可以调用外部程序,通常可以用来作多进程的工具,这里我们用 call 方法来调用 ping 命令,其中的三个参数,shell=True 表示接收字符串形式的命令,为了不在终端显示 ping 命令的输出,通过 stdout 和 stderr 将标准输出和标准错误输出进行重定向。

Queue 的作用是用来创建一个单项队列,通过这个队列管理线程会更方便一些,其中用到的几个方法已经做了注释说明,看程序就能明白作用了。

最后运行的效果类似这样。

```
Main Thread waiting...
Thread 0 : Pinging: 10.0.2.11
Thread 1 : Pinging: 10.0.2.12
Thread 2 : Pinging: 10.0.2.13

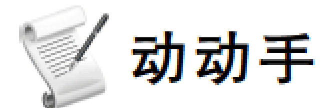
10.0.2.12 : is alive
Thread 1 : Pinging: 10.0.2.14
10.0.2.11 : is alive
Thread 0 : Pinging: 10.0.2.15
10.0.2.13 : did not respond
10.0.2.14 : is alive
10.0.2.15 : is alive
Done
```

重点提示

在这一章中，你要掌握的内容：

(1) 明确进程和线程的作用和区别。

(2) 掌握多线程的创建和控制方法。



动动手

实现一个多线程备份大文件的类，带有智能判断功能，所有大于指定大小的文件自动进行多线程备份，比如有 10 个日志，每个 1GB，则实现多线程备份。

第 14 章 网络应用编程

在互联网时代，编程又怎能离开网络呢！Python 标准库中提供了丰富的网络相关功能的支持，总的来说，分成两类：一类是针对特定应用级的网络协议封装的库，比如 FTP、HTTP、SMTP、POP 等；另一类是提供了底层操作系统中的基本套接字(socket)支持，可以实现面向连接和无连接协议的客户端和服务器。

除了标准库以外，还有专门针对网络编程的框架 Twisted。Twisted 是用 Python 实现的基于事件驱动的网络引擎框架，Twisted 支持许多常见的传输及应用层协议，包括 TCP、UDP、SSL/TLS、HTTP、IMAP、SSH、IRC 以及 FTP。

如果你的网络基础知识不好，也不用担心，我们从应用出发来了解用 Python 如何进行网络编程。

14.1 网络应用开发

Python 标准库中的网络模块非常好用，即使你不明白网络协议是什么也可以很轻松地实现网络应用的基本开发，下面我们来看一下 urllib 库的基本用法。

首先从名字就能看出来，urllib 这个库主要是用来处理 URL(统一资源定位符)的。

通过 urllib 库可以很方便地获取指定网页的源代码，如果使用 GUI 的 HTML 组件，就可以轻松地做一个简单的 Python 实现的 Web 浏览器。

urllib 库包括以下模块。

(1) urllib. request：请求模块。

(2) urllib. error：异常处理模块。

(3) urllib. parse url：解析模块。

(4) urllib. robotparser：robots. txt 解析模块。

下面介绍怎么用 urllib. request 获取一个网页的源代码，这个功能对于写爬虫非常实用。

urllib. request 的 urlopen 方法可以返回一个类似 file 的对象，可以像读取文件一样获取网页源代码。就像下面这样：

```
>>>import urllib.request
>>>page = urllib.request.urlopen("http://www.python.org")
>>>page.readline()
b'<!doctype html>\n'
```

这是最简单的形式，只需要提供一个 URL 地址给 urlopen 即可，page 获得的值就是当前网页的源代码。

如果想把源代码保存为一个文件，可以通过 urllib. request 的 urlretrieve 方法实现：

```
>>>from urllib import request
>>>request.urlretrieve("http://www.python.org","file.txt")
('file.txt', <http.client.HTTPMessage object at 0x06639890>)
>>>import os
>>>os.listdir()
['boa-constructor-wininst.log', 'cla-fs.py', 'DLLs', 'Doc', 'file.txt', 'hero.db',
'include', 'NEWS.txt', 'python.exe']
```

除了 urllib，还有一些看名字就知道用处的模块，如果有需要可以在手册查找，通常手册上都会有一些简单的例子，很实用。

（1）ftplib：用于访问 FTP 服务器，可上传、下载等。

（2）poplib：实现了 POP 3 协议，用来收邮件。

（3）smtplib：实现 SMTP 协议，可以发送纯文本邮件、HTML 邮件以及带附件的邮件。

14.2 socket 编程



网络编程 socket

对于 Python 网络编程，一定会用到 socket 模块。socket（套接字）是双向通信信道的端点。应用程序通常通过“套接字”向网络发出请求或者应答网络请求，使主机间或者一台计算机上的进程间可以通信。在不同主机的进程之间进行通信，主机可以是任何一台连接网络的机器。

我们可以通过 socket 模块编写客户端和服务器端程序，比如编写局域网聊天室（或者聊天软件）、文件传输工具程序等。

前面介绍的封装好的模块，用起来很方便，但是做底层开发就不够灵活了，而 socket 模块则提供了更底层的接口。

如果以创建一个游乐园为例，封装好的模块相当于整套过山车或一个精灵屋，你可以拿来用，但是不能做更多更灵活的更改，比如，不能把过山车变成精灵屋，因为它的功能已经被限定了。而 socket 模块则像螺丝、零件、水泥等，你想要什么，就可以灵活地建造。

14.2.1 socket 连接过程

网络服务都是建立在 socket 基础之上的，socket 是网络连接端点，是网络的基础，每个 socket 都被绑定到指定的 IP 和端口上。

套接字可以通过多种不同的通道类型实现，如 Unix 域套接字、TCP、UDP 等。套接字库提供了处理公共传输的特定类，以及一个用于处理其余部分的通用接口。

利用 socket 建立网络连接的步骤（一对套接字连接过程）如下。

（1）服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

（2）客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接

字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后向服务器端套接字提出连接请求。

(3) 连接确认：当服务器端套接字监听到或者接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

14.2.2 创建 socket 对象

了解了 socket 的连接步骤，我们就可以创建 socket 了。首先，无论是服务端还是客户端都需要先生成一个 socket 对象。

socket 是进程通信的一种方式，即调用这个网络库的一些 API 函数实现分布在不同主机的相关进程之间的数据交换。那么要想在两个主机之间建立连接，就要遵守同样的规则。这里有些概念需要先了解。

(1) 每个 socket 都被绑定到指定的 IP 和端口上。

① IP 地址：即依照 TCP/IP 协议分配给本地主机的网络地址，两个进程要通信，任一进程首先要知道通信对方的位置，即对方的 IP。

② 端口号：用来辨别本地通信进程，一个本地的进程在通信时均会占用一个端口号，不同的进程端口号不同，因此在通信前必须要分配一个没有被访问的端口号。

(2) 数据传输的模式基本是 TCP 和 UDP 两种，TCP 和 UDP 的区别如下。

① TCP 是面向连接的，连接经过三次握手，很大程度保证了连接的可靠性。

② UDP 传送数据前并不与对方建立连接，对收到的数据也不发送触诊信号，因此 UDP 的开销更小，数据传输速率更高。QQ 就是采用 UPD 协议传输，而相似的 MSN 采用的是 TCP 协议传输。

创建 socket 对象需要通过 socket 模块的 socket 方法。语法如下：

```
socket.socket([family[, type[,protocol]]])
```

参数说明如下。

(1) family 地址簇：用于 socket() 函数的第一个参数，有以下三个模式。

① socket. AF_UNIX：用于单一机器下的进程通信。

② socket. AF_INET：用于服务器之间相互通信(通常都用这个)。

③ socket. AF_INET6：支持 IPv6。

(2) type：两个端点之间的通信类型。

① 用于 TCP 的面向连接的协议的 SOCK_STREAM。

② 用于 UDP 的无连接协议的 SOCK_DGRAM。

(3) protocol：通常为 0，用于标识域和类型中的协议的变体，可选项，不用指定。

创建 TCP socket：

```
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```



创建 UDP socket：

```
s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

14.2.3　基于 TCP 的客户端和服务端

socket 对象的方法按用处可以分为三类：服务端、客户端和通用。

以 TCP 为例，分别创建服务端和客户端的基本模型，由客户端向服务端发送字符串，服务端接收并打印到屏幕上，然后发给客户端一个字符串。

服务端代码如下：

```
#tcpserver.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1", 6000))
s.listen(10)
print('Waiting for connection...')

clients, clientaddr = s.accept()
data = clients.recv(1024)
print("client %s say: %s" % (clientaddr, data.decode()))

clients.send(b"hello, i am Server")
clients.close()
```

客户端代码如下：

```
#tcpclient.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 6000))
s.send(b"hello,i am client")

data = s.recv(1024)
print("server say: %s" %data.decode())
s.close()
```

因为这里的两个脚本都在一台主机上，所以在运行时直接在命令行模式用两个终端分别运行。先启动一个终端运行服务端程序：

```
C:\CrazyPythonZeroToHero\code\14>python tcpserver.py
Waiting for connection...
```

此时服务器处于监听状态，也就是等待有人来连接，接下来再启动一个命令行模式终端运行客户端，当客户端运行时，就会向服务器发送消息，服务器也会监听到连接，然后继续后面的操作。

客户端运行后的效果：

```
C:\CrazyPythonZeroToHero\code\14>python tcpserver.py
server say: b'hello, i am Server'
```

接收到客户端信息之后的服务端：

```
C:\CrazyPythonZeroToHero\code\14>python tcpserver.py
Waiting for connection...
client ('127.0.0.1', 50759) say: b'hello,i am client'
```

看到效果后我们来解释一下服务端和客户端的关键代码。

1. 服务端

(1)
```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

创建 socket 对象，同时通过参数设定 socket 对象的通信模式为 AF_INET，数据传输模式为 TCP。

(2)
```
s.bind(("127.0.0.1", 6000))
```

绑定服务器的 IP 地址和端口，如果地址为空则表示本机，地址最好大于 5000 小于 65535，因为预留端口就在这个区间。

(3)
```
s.listen(backlog)
```

开始 TCP 监听。backlog 指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为 1，大部分应用程序设为 5 就可以了。

(4)
```
clients, clientaddr = s.accept()
```

被动接受 TCP 客户端连接，(阻塞式)等待连接的到来，接收来自客户端的数据，返回一个新的 socket 对象和客户端地址。

(5)
```
data = clients.recv(bufsize)
```

接收 TCP 数据，数据以字符串形式返回，bufsize 指定要接收的最大数据缓存量。

(6)
```
clients.send(b"hello, i am Server"))
```

向已经连接的 socket 发送数据，在 Python 3 中，所有数据的传输必须用 bytes 类型(bytes 只支持 ascii 码)，所以在发送数据时要么在发送的字符串前面加 'b'，要么使用 encode('utf-8')转换成 bytes 类型发送，但是在接收端必须用 decode()进行转码。

2. 客户端

```
s.connect(("127.0.0.1", 6000))
```



客户端跟服务端不同，不需要绑定也不需要监听，但是需要制定连接的服务器，connect 连接到给定地址处的套接字。一般地址的格式为元组(hostname，port)，如果连接出错，返回 socket.error 错误。

至此，一对简单的客户端和服务端就实现了。你可以考虑写一个服务端和客户端来回发消息的聊天工具，写完检查是否有问题，有没有改进的办法。然后再考虑能不能实现一个带图形界面的局域网聊天室。

14.2.4 基于 UDP 实现多线程收发消息

14.2.3 小节的最后我们说到了要实现服务端和客户端互发消息，如果你动手做了，并且采取 TCP 协议直接将 accept、send 和 recv 放到了循环中，就会发现一个问题，这样简单的循环是无法实现一直发消息或者一直接收消息的，因为 accept 方法会产生中断的效果，并且 send 和 recv 又是按顺序执行的。要怎么解决这个问题呢？聪明的你想到多线程没有？

不错，我们只需要将发送和接收分别作为两个线程，这样就不会互相影响了。本小节我们通过 UDP 以及多线程来实现这个例子，掌握方法后你也可以试一下 TCP 协议的服务端和客户端模式来实现。

首先要说明一下 UDP 的工作方式跟 TCP 有一个区别：UDP 模式不需要 accpet 等待来自客户端的连接请求，发送端只需要指定地址和端口直接发送，接收端则需要绑定字节接收信息的地址和端口。下面就来看下基本的发送端和接收端代码。

```
#发送端
#udpSend.py
import socket
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    addr = ("192.168.122.1", 6000)
    s.sendto("Hello Python", addr)   #sendto 的作用是将字符串发送至指定地址

#接收端
#udpRecv.py
import socket
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    addr = ("192.168.122.1", 6000)   #本机 IP 地址和端口
    s.bind(addr)
    data, addr = s.recvfrom(1024)    #1024 是缓存数据,单位是 bytes
    print(data)
```

明白了 UDP 的工作方式，就可以结合多线程来实现类似聊天软件的收发消息功能了。因为可以通过 sento 向任意主机发送消息且不需要对方 accept，所以可以用一个通用脚本运行在各个主机上就可以了，代码如下：

```
#sock_udp_thread.py
import socket
from threading import Thread
```

```
PORT = 6000          #所有终端使用同一个端口
HOST = '192.168.122.1' #本机 IP
def send(sock, addr):
    i_say = ''
    while i_say != 'bye':
        i_say = input('i say:')
        sock.sendto(i_say.encode('utf8'),addr)

def recv(sock):
    while True:
        pass #event ctrl
        other_say, addr = sock.recvfrom(1024)
        print('{} say>{}'.format(addr[0],
                                other_say.decode('utf-8')))

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind((HOST, PORT))
    ipaddr = input('you want say to (ip):')  #获取接收者的 IP
    addr = (ipaddr, 6000)

    sendTH = Thread(target = send, args = (s, addr))
    sendTH.setDaemon(True)
    sendTH.start()
    recvTH = Thread(target = recv, args = (s,))
    recvTH.setDaemon(True)
    recvTH.start()
    sendTH.join()
    recvTH.join()
```

这样每个主机都运行这个脚本，就可以向其他主机发送消息了。如果你是在一台主机做这个实验，可以用两个终端运行两个绑定了本机不同 IP 地址的脚本来做实验，比如我就是一个绑定到 127.0.0.1，另一个绑定到 192.168.122.1。

14.3 实例：局域网聊天室

掌握了 GUI、多线程等很多实用的工具后，我们来写一个局域网聊天室吧。之前我们已经接触到了一些软件工程中的步骤，比如需求分析、编码、测试等，但是并不系统，所以就以软件工程开发的形式来开发这个聊天室。

14.3.1 需求分析

开发软件的第一步是需求分析，就像我们在写《英雄无敌》时，就进行了一些初步的分析。实际工作中，通常会是产品经理、客户、测试工程师等项目相关人员讨论软件的具体功能，以及大概的效果等，最后会生成软件需求说明书。程序员拿到需求说明书就可以按步骤开发了。

现在对局域网聊天室的功能需求主要有以下几点。

（1）软件支持多人同时在局域网内首发文字消息，每个人发送的消息，大家都能看到。

（2）每个人可以给自己起个昵称，但昵称不能为空，软件启动时，在用户未起昵称前，随机生成一个 4 个字符的昵称。

（3）聊天记录中显示聊天信息的格式为用户昵称-时间-换行-用户消息。

（4）用户启动软件后，自动加入聊天室。

14.3.2　概要设计

确认了需求之后，第二步是进行概要设计，这时的工作就是将软件按照功能进行划分，分多个模块，每个模块实现独立功能，各个模块提供互相配合的接口。作为图形工具，还需要完成聊天室的界面设计工作。

根据图 14-1 这个界面和需求分析，局域网聊天室项目主要分为三个模块。



图 14-1　局域网聊天室原型

（1）GUI 模块：负责接收用户的消息并显示所有人发出的消息，提供一个良好的交互体验。

（2）消息接收模块：负责从网络接收用户发出的消息。

（3）消息发送模块：将用户的消息发送出去。

14.3.3　详细设计

概要设计只是将任务划分开，接下来就是每个模块内部具体的实现方式。如果是比较大的项目，每个模块分给一个项目组，各个项目组要针对各自的模块功能进行更具体的设计，比如模块实现的流程、具体的功能实现方式等。三个模块具体的流程分别设计如下。

（1）用户界面模块流程，见图 14-2。

（2）消息发送模块流程，见图 14-3。

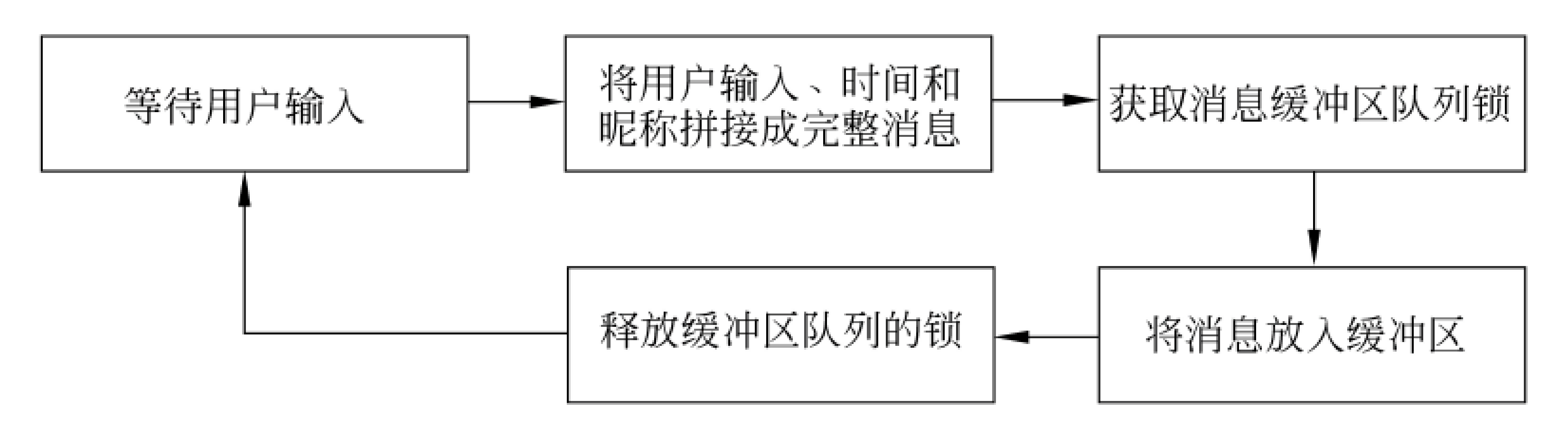


图 14-2　用户界面模块流程图

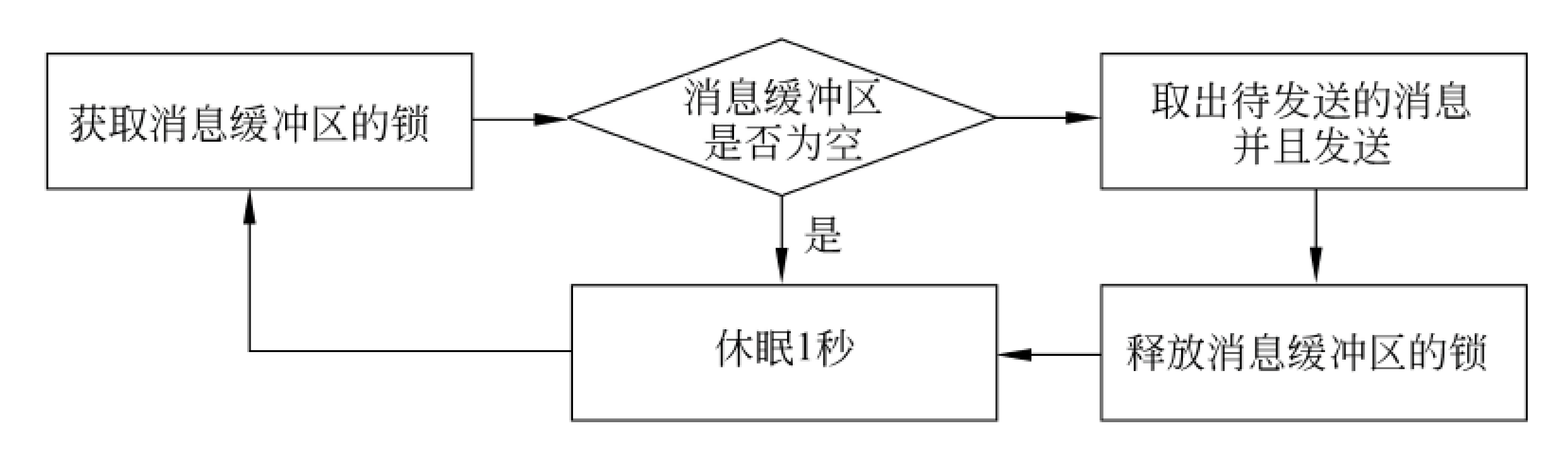


图 14-3　消息发送模块流程图

（3）消息接收模块流程，见图 14-4。

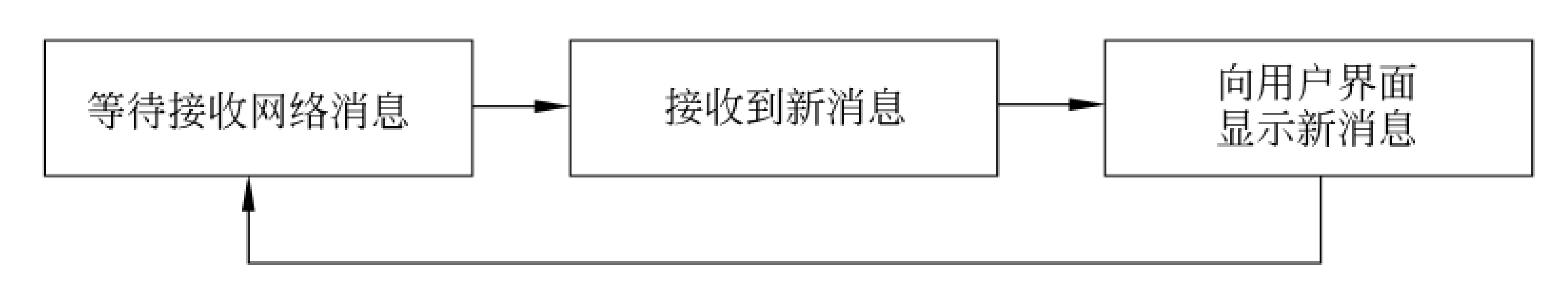


图 14-4　消息接收模块流程图

分别设计出模块后，可以看出，每个模块都可以单独工作，但又需要其他模块的消息，比如 GUI 要一直在屏幕上显示，同时有人发消息，又都会接收到消息。三个模块需要同时工作，所以，在程序设计时要用到多线程。另外，在消息处理的时候也需要多线程，前面实现的简单例子是单线程的，客户端和服务端必须等待上一个操作完成才会进入下一步，这显然不符合聊天室的模式(你不能在发消息时还要先等着别人的消息到了再发)。

综合三个模块来看一下数据从输入到发送再到接收的完整数据流，如图 14-5 所示。

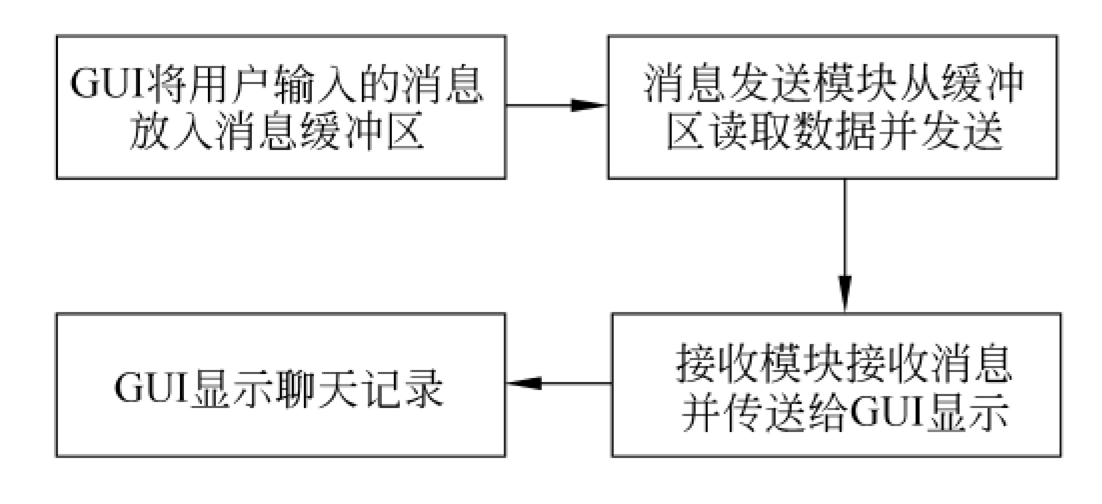


图 14-5　模块间数据流

从图中会发现一点，GUI 向消息缓冲区放入数据，消息发送模块从消息缓冲区读取数据并发送，消息接收模块接收消息并传送给 GUI 显示，两个模块同时运行并且使用同一个消息缓冲区，所以，在消息缓冲区读写时需要加锁，以保证同一时间段只有一个线程。

14.3.4 编码阶段

当完成详细设计之后就可以将所有工作交给程序员了,程序员要做的就是将所有设计描述编成代码来实现。这里分三个模块,假设将三个模块给了三个小组,每个小组完成自己的部分,最终合在一起运行。

我们先大致了解一下开发的想法,然后再看代码。首先三个模块分别作为三个类来开发:GUI 模块类、接收消息模块类、发送消息模块类。最后合并时可以把三个模块分别存储为一个文件,然后在一个逻辑代码中导入这些模块,下面的做法是把三个模块中的类直接合并到一个程序文件 MyChart0.1.py 中。

通过 wxPython 实现 GUI 设计。socket 编程部分,因为在前面介绍了 socket 的 TCP 传输,为了多了解一些,所以这里用 UDP 来实现,主要是涉及的函数方法不同。最后,用组播地址的办法实现消息发送(组播地址的作用是,你向这个地址发送消息,数据会发到局域网内的每台主机,这样就不必知道每台主机的地址,再一个一个发送了),所以,你也会发现我们的程序只有一个文件,并没有分成服务端和客户端。

下面是整个程序各个组成部分的代码,因为要做讲解,所以做了比较详细的注释,实际的程序不需要这么详细,做好关键注释就好。

(1) 将要用到的模块依次导入,主要是 wxPython、threading、socket 等。

```
#MyChart0.1.py
import wx
import socket
import threading
import queue
import time
import random
import string
```

(2) 用户界面模块,也就是 wxPython 实现的代码。

```
class MyChartFrame(wx.Frame):
    """GUI 模块"""
    def __init__(self, parent, title, size = (640,480)):
        """构造函数,初始化主要方法"""
        wx.Frame.__init__(self, parent, title = title, size = (640,480))

        #等待发送的消息的队列
        self.sendMsgQueue = queue.Queue(200)
        #生成将要给消息队列上的锁,用于保证同一时刻只有一个线程对队列操作
        self.sendMsgQueueLock = threading.Lock()

        self.initUI()
        self.bindEvent()
        self.startMsgThread()
        self.Show()
```

```
    def initUI(self):
        seed = "0123456789"
        randomName = "".join(random.sample(seed, 8))

        panel = wx.Panel(self)
        vbox = wx.BoxSizer(wx.VERTICAL)                 #全局布局管理器

        #聊天记录组件及布局
        self.tcMsgAll = wx.TextCtrl(panel,
                                    style = wx.TE_READONLY|wx.TE_MULTILINE,
                                    size = (530, 300))
        vbox.Add((-1,15))                               #增加窗口中的空白空间
        vbox.Add(self.tcMsgAll,proportion = 1,
                flag = wx.EXPAND|wx.LEFT|wx.RIGHT, border = 10)

    #昵称组件及布局
        stName = wx.StaticText(panel, label = '昵称')
        self.tcName = wx.TextCtrl(panel)
        self.tcName.SetValue(randomName)
        hboxName = wx.BoxSizer(wx.HORIZONTAL)           #昵称行管理器
        hboxName.Add(stName,
                  flag = wx.RIGHT|wx.ALIGN_CENTER_VERTICAL,border = 10)
        hboxName.Add(self.tcName,
                  flag = wx.ALIGN_CENTER_VERTICAL)
        vbox.Add(hboxName,
                flag = wx.EXPAND|wx.LEFT|wx.RIGHT|wx.TOP, border = 20)
        vbox.Add((-1,15))                               #增加窗口中的空白空间

        #用户输入内容的文本框和发送消息的按钮
        self.tcMsgEnt = wx.TextCtrl(panel, style = wx.TE_MULTILINE, size = (500, 80))
        self.btnSendMsg = wx.Button(panel, label = "发送")
        hboxMsgEnt = wx.BoxSizer(wx.HORIZONTAL)
        hboxMsgEnt.Add(self.tcMsgEnt, proportion = 1,
                   flag = wx.EXPAND|wx.ALIGN_CENTER_VERTICAL|wx.RIGHT, border = 10)
        hboxMsgEnt.Add(self.btnSendMsg,
                   flag = wx.EXPAND|wx.ALIGN_CENTER_VERTICAL|wx.RIGHT, border = 5)
        vbox.Add(hboxMsgEnt, flag = wx.EXPAND|wx.LEFT|wx.RIGHT, border = 15)
        vbox.Add((-1,15))

        panel.SetSizer(vbox)                            #启动布局管理器
        vbox.SetSizeHints(self)                         #组件最小限制

    def btnEvtSendMsg(self,evt):
        """按钮事件处理"""
        n = self.tcName.GetValue()
        t = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(time.time()))
        m = self.tcMsgEnt.GetValue()
        msg = "%s %s :\n%s\n\n" % (n, t, m )
        self.tcMsgEnt.Clear()
```

```
        self.sendMsgQueueLock.acquire()           #获取消息队列锁
        self.sendMsgQueue.put(msg.encode('utf8'))#向队列放入数据
        self.sendMsgQueueLock.release()           #释放消息队列锁

    def bindEvent(self):
        """绑定发送按钮事件"""
        self.btnSendMsg.Bind(wx.EVT_BUTTON, self.btnEvtSendMsg)

    def startMsgThread(self):
        """实例化接收和发送消息的模块,并启动线程"""
        self.recvMsgThread = RecvMsgThread(self)
        self.sendMsgThread = SendMsgThread(self.sendMsgQueue, self.sendMsgQueueLock)
        self.recvMsgThread.start()
        self.sendMsgThread.start()

    def RefreshTcMsgAll(self, msg):
        """刷新聊天记录"""
        self.tcMsgAll.AppendText(msg)

    def __del__(self):
        """析构函数,程序结束时,结束线程"""
        self.recvMsgThread.stop()
        self.sendMsgThread.stop()
        self.recvMsgThread.join()
        self.sendMsgThread.join()
```

在这个类中,我们跟另外两个类相关联的就是通过 startMsgThread 将另外两个类实例化为线程。因为收发消息都会对队列数据进行操作,所以在收发数据时都要对队列加锁。

(3)发送消息的代码。

```
class SendMsgThread(threading.Thread):
    """消息发送模块"""
    def __init__(self, sendMsgQueue, sendMsgQueueLock):
        threading.Thread.__init__(self)
        self.threadName = "SendMsgThread"
        self.exitFlag = threading.Event()

        self.sendMsgQueue = sendMsgQueue
        self.sendMsgQueueLock = sendMsgQueueLock

    def run(self):
        group_ip = "224.1.1.1"                  #组播地址
        port = 8000                             #自定义端口
        print("starting ", self.threadName)     #在后台终端输出

        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
        s.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 32)
```

```
        #监视消息队列,一旦有消息就读取消息并发送,直到线程停止
        while not self.stopped():
            self.sendMsgQueueLock.acquire()         #获取队列锁
            if not self.sendMsgQueue.empty():
                msg = self.sendMsgQueue.get()       #读取队列消息
                self.sendMsgQueueLock.release()     #取得消息后释放队列锁
                s.sendto(msg, (group_ip, port))
                print("%s sending %s" % (self.threadName, msg))
            else:
                self.sendMsgQueueLock.release()
            time.sleep(1)

        s.close()
        print("Exiting ", self.threadName)

    def stop(self):
        self.exitFlag.set()

    def stopped(self):
        return self.exitFlag.isSet()
```

发送消息用到了组播地址 224.1.1.1,组播地址的工作方式是将所有的信息接收者都加入一个组内,并且一旦加入之后,流向组地址的数据立即开始向接收者传输,组中的所有成员都能接收到数据包。组播组中的成员是动态的,主机可以在任何时刻加入和离开组播组。这里的 224.1.1.1 就相当于向局域网的所有主机发送消息。

self.exitFlag = threading.Event():threading.Event 机制类似于一个线程向其他多个线程发号施令的模式,其他线程都会持有一个 threading.Event 的对象,这些线程都会等待这个事件的"发生",如果此事件一直不发生,那么这些线程将会阻塞,直至事件的"发生"。

(4) 消息接收模块。

```
class RecvMsgThread(threading.Thread):
    """消息接收模块"""
    def __init__(self, mainThread):
        threading.Thread.__init__(self)
        self.threadName = "RecvMsgThread"
        self.mainThread = mainThread
        self.exitFlag = threading.Event()

    def run(self):
        group_ip = "224.0.0.1"
        port = 8000

        print("starting ", self.threadName)
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
        s.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 32)
```

```
        s.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_LOOP, 1)

        s.bind(('', port))
        host = socket.gethostbyname(socket.gethostname())
        s.setsockopt(socket.SOL_IP, socket.IP_MULTICAST_IF, socket.inet_aton(host))
        s.setsockopt(socket.SOL_IP, socket.IP_ADD_MEMBERSHIP,
                    socket.inet_aton(group_ip) + socket.inet_aton(host))

        #从网络接收消息,接收到消息便显示,直到线程被停止
        while not self.stopped():
            try:
                msg, addr = s.recvfrom(1024)
                wx.CallAfter(self.mainThread.RefreshTcMsgAll, msg)
            except socket.error as e:
                print("receiving expection")
            time.sleep(1)

        s.close()
        print("Exiting ", self.threadName)

    def stop(self):
        self.exitFlag.set()

    def stopped(self):
        return self.exitFlag.isSet()
```

这个模块主要的功能就是 socket 对象会循环的从 8000 端口接收消息,然后通过 wx.CallAfter(self.mainThread.RefreshTcMsgAll, msg)调用 GUI 模块的 RefreshTcMsgAll 方法显示消息。

(5) 最后的逻辑代码只要启动 GUI 就可以了。

```
if __name__ == "__main__":
    app = wx.App()
    frame = MyChartFrame(None, title = 'Milo 的局域网聊天室 V0.1')
    frame.Show(True)
    app.MainLoop()
```

至此,你的局域网聊天室就可以使用了。如果你是重新写的这段代码,很可能会有编写错误、拼写错误等问题,慢慢来,注意看报错信息。最终效果如图 14-6 所示。

这个聊天室只是聊天功能其中的一种实现方式,需求不同就会产生不同的效果,比如你还可以加上文件传输的功能,也可以设计一个服务器端,负责跟各个客户端连接,客户端可以指定昵称来收发消息等。

重点提示

在这一章中,你要掌握的内容:

(1) 理解 urllib 等网络应用模块的用法。



图 14-6　运行效果图

（2）理解 socket 编程的核心原理。

（3）能独立设计客户端/服务端模式的程序。



动动手

（1）亲自动手完成本章的局域网聊天室实例，深入理解后，如果条件允许（比如你懂虚拟机，或者你有局域网）可以将聊天室改成类似局域网即时通信工具，用户可以指定昵称或 IP 进行聊天。

（2）看手册及手册案例了解 socketserver，如果不太明白，扫描二维码看这两个视频吧：看手册了解 socketserver 和一个压缩网络数据的小案例。



看手册了解 socketserver



压缩网络数据

第 15 章 正则表达式

正则表达式(Regular Expression)又叫正规表示法，主要用于对字符串进行操作。正则表达式并不是 Python 特有的，而是计算机科学的一个概念，很多语言都支持正则表达式。


正则表达式

在编程过程中，处理最多的数据类型就是字符串。正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

很多时候都需要用到正则表达式。比如，你想通过程序判断一个字符串是否是一个合格的 E-mail 地址，或者写爬虫获得了一个网页的源代码，然后要从大篇源代码过滤出有用的数据。是不是想想都头疼？但这些问题用正则表达式就可以轻松解决。

15.1 正则表达式的常用字符

如果你接触过正则表达式一定会见过一些看起来很奇怪的字符组合，比如“.+？@.+\.+”等。在正式介绍正则表达式之前，还要明确一点，通常正则表达式就是一个特殊的字符串，通过工具判断某个待判断字符串是否符合正则表达式字符串所代表的含义。所以接下来要了解有哪些字符用来组成正则表达式。

在正则表达式中，分为普通字符和元字符两种。

如果你是第一次接触正则表达式，可以先利用一些比较方便的方法适应一下，比如这个在线的网站 http://rubular.com/(见图 15-1)。

在这个在线网站中直接输入正则表达式和待匹配字符串就可以看到匹配结果。这对于刚开始熟悉正则表达式，或者想快速验证正则表达式，非常方便。

15.1.1 普通字符

普通字符就是仅代表自己，没有特殊含义，比如字母和数字。如果正则表达式写成 abc，则只有 abc 这个字符串符合这个正则的要求。比如字符串 vdgpythonsha，我想知道这一串字符串中是否有 python 这个单词，那么，就可以使用 python 作为一个正则表达式，然后将 vdgpythonsha 与我这个正则表达式进行相应的匹配，结果是其中含有 python。这时的正则表达式就都是普通字符。



图 15-1　http://rubular.com/网站

但有时可能要做一些模糊匹配，比如 shjPythoniipython 这个字符串如果用 python 来匹配，因为严格区分大小写的原因，可能只匹配到一个（说“可能”，是因为正则有办法不区分大小写，下面将详细介绍）。

15.1.2　元字符

普通字符只能进行严格死板的匹配。在正则表达式中可以通过元字符来完成更复杂的匹配，元字符是正则表达式中具有特定含义的字符，常用元字符及功能如表 15-1 所示。你可以把正则和待匹配的字符串放到 http://rubular.com/测试一下，看看直观的效果。

表 15-1　元字符

元字符	含　　义	正　　则	匹　　配
.	匹配任意除换行符“\n”外的字符 (在 DOTALL 模式中也能匹配换行符)	a.c	abc、axc
\	转义字符，使后一个字符改变原来的意思，如果是元字符则变成普通字符	a\.c	a.c
*	匹配前一个字符 0 或多次	abc*	ab-abccc
+	匹配前一个字符 1 次或无限次	abc+	abc-abccc
?	匹配前一个字符 0 次或 1 次	abc?	ab;abc
^	匹配字符串开头。在多行模式中匹配每一行的开头	^abc	abcxyz
$	匹配字符串末尾，在多行模式中匹配每一行的末尾	abc$	xyzabc
|	或。匹配“|”左右表达式任意一个，从左到右匹配，如果“|”没有包括在()中，则它的范围是整个正则表达式	abc|def	abc、def

续表

元字符	含　　义	正　　则	匹　　配
{}	{m}匹配前一个字符 m 次;{m,n}匹配前一个字符 m 至 n 次,若省略 n,则匹配 m 至无限次	ab{1,2}c	abc、abbc
[]	字符集。对应的位置可以是字符集中任意字符。[^abc]表示取反,即非 abc。所有特殊字符在字符集中都失去其原有的特殊含义。用“\”反斜杠转义恢复特殊字符的特殊含义	a[bcd]e	abe、ace、ade
()	被括起来的表达式将作为分组,从表达式左边开始每遇到一个分组的左括号“(”,编号+1 分组表达式作为一个整体,可以后接数量词。表达式中的“|”仅在该组中有效。分组等用法很多,可以通过 help(re)查看详细说明	(abc){2} a(123|456)c	abcabc a456c

元字符除了单个使用,还可以组合在一起实现更复杂的匹配,比如我们要匹配一个手机号码是不是移动 13 开头号段,就需要组合几个元字符。

首先限定了 13 号段,那么就以“13”这个普通字符开始。第三位移动的是从“4”到“9”中的一个,就可以用[4-9]来表示,手机号一共是 11 位,从第四位到第十一位的 8 个数字可以是“0”到“9”的任意数字组合,可以用[0-9]{8}表示。这样组合到一起作为一个正则表达式,如图 15-2 所示。

```
"13[4-9][0-9]{8}"
```

图 15-2　效果图

还有一个问题就是,如果待匹配的字符串不是“13”开头,比如 a13512345678x 这样的字符串,其中有符合的部分,也会匹配成功,所以可以加上限制开头的“^”,这样可以限定为以 13 开头。同理,11 位号码之后还有其他字符,比如字符串 13512345678abc 也是可以匹配的,所以,再加上“$”限制结尾,这样就能完全匹配 11 位了,最终结果如图 15-3 所示。

```
"^13[4-9][0-9]{8}$"
```

匹配数字相对简单一些,假设要验证一个字符串是否是一个合法的 E-mail 地址,就要稍复杂一点。

还是要先确定 E-mail 的特征,虽然没有统一的邮箱账号格式,但是所有邮箱都符合“名称@域名”的规律。我们主要是根据项目需要进行匹配,毕竟现在已经有邮箱支持中文的 E-mail 地址了。

例:只允许英文字母、数字、下划线、英文句号、以及中划线组成。如 zouqixian@gmail.com 或 zouqixian-01@gmail.com.cn。



图 15-3　匹配对比图

(1) 名称部分

26 个大小写英文字母、数字、下划线、中划线表达式为[a-zA-Z0-9_-]。

由于名称是由若干个字母、数字、下划线和中划线组成，所以需要用到＋表示多次出现，得出邮件名称表达式为[a-zA-Z0-9_-]＋。

(2) 域名部分

一般域名的规律为[N 级域名][三级域名.]二级域名.顶级域名，比如 gmail.com、gmail..com.cn，分析可得域名类似"*.*.*.*"组成。

"**"部分可以表示为[a-zA-Z0-9]＋。

".**"部分可以表示为\.[a-zA-Z0-9]＋。

多个".**"可以表示为(\.[a-zA-Z0-9]＋)＋。

综上所述，域名部分可以表示为[a-zA-Z0-9_-]＋(\.[a-zA-Z0-9_-]＋)＋。

由于邮箱的基本格式为"名称@域名"，需要使用"^"匹配邮箱的开始部分，用"$"匹配邮箱结束部分以保证邮箱前后不能有其他字符，所以最终邮箱的正则表达式如下：

```
"^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$"
```

你可能发现经常出现类似 a-z、0-9 这样的字符集，所以，正则中有一些有用的预定义字符集，举例如下。

\d：匹配一个数字，等同于[0-9]。

\D：匹配非数字，等同于[^0-9]。

\s：匹配空白，如果带有 re.ASCII，则匹配\t\n\r\f\v 中的一个。

\S：匹配非空白。

\w：匹配大小写字母、数字和下划线，等同于[a-zA-Z0-9_](注意包含下划线)。

\W：匹配 Unicode 非大小写字母、数字和下划线，等同于[^a-zA-Z0-9_]。

这样一来，上面的正则可以简化为

```
"^[\w-]+@[\w-]+(\.[\w-]+)+$"
```



如果是中文的名字怎么匹配呢？其实很简单，匹配中文字符的正则表达式：[u4e00-u9fa5]，如果需要匹配 2 到 4 位的中文就可以写成[u4e00-u9fa5]{2,4}。

正则表达式还有很多规则和用法，这里我们仅通过一些演示来让你快速入门。正则其实并不难，了解每个符号的意思后，自己动手试一试，多写几次就明白了，正则是出了名的坑多，少写了个点就匹配不到数据了，多练多试，慢慢就好了。

15.2 Python 中的 re 模块

大致了解了正则表达式的写法和规则后，如何在 Python 中通过正则表达式处理字符串呢？

Python 中提供了一个 re 模块，用来支持正则表达式，re 模块中提供了丰富的函数和相应的参数。

15.2.1 正则表达式主要功能

正则表达式处理字符串主要有以下四大功能。

(1) 匹配：查看一个字符串是否符合正则表达式的语法，一般返回 true 或者 false。

(2) 获取：正则表达式来提取字符串中符合要求的文本。

(3) 替换：查找字符串中符合正则表达式的文本，并用相应的字符串替换。

(4) 分割：使用正则表达式对字符串进行分割。

15.2.2 re 模块使用的两种形式

Python 中的 re 模块使用正则表达式有两种形式。

一种是直接通过模块调用函数进行操作，通常第一个参数是正则表达式字符串，然后是待匹配的内容，此种方法适合只使用一次的正则表达式。比如：

```
>>>re.findall(r"^13[4-9][0-9]{8}$","13512345678")
```

另一种是使用 re.compile(r, f)方法生成正则表达式对象，然后调用正则表达式对象的相应方法。这种做法的好处是生成正则对象之后可以多次使用。比如：

```
>>>rx = re.compile(r"^13[4-9][0-9]{8}$")
>>>rx.findall("13512345678")
```

可能你注意到了定义正则字符串前的 r，需要说明一下。正则表达式通过“\”作为转义字符，比如“\n”表示换行，但有时我们就是要匹配“\n”这两个字符，你可以这样表示“\\n”，如果有多个类似的符号就会看起来比较乱，也不易读，所以，这时可以在字符串前加 r，代表这是一个原生字符串，其中的转义字符不转义。

15.2.3 re 常用函数及方法

re 模块包含的内容非常多，建议你通过 help(re)查看详细说明。

1. re.compile()

编译正则表达式，返回一个对象的模式（可以把常用的正则表达式编译成正则表达式对象，提高效率）。

函数语法为

```
re.compile(pattern,flags = 0)
```

pattern：编译时用的表达式字符串。

flags：编译标志位，用于修改正则表达式的匹配方式，如是否区分大小写、多行匹配等。flags 在其他函数中都是通用的，我们分开在几个函数中举例，常用的 flags 见表 15-2 所示。

表 15-2　常用的 flags

标　志	含　义
re. S(DOTALL)	使匹配包括换行在内的所有字符
re. I(IGNORECASE)	使匹配对大小写不敏感
re. L(LOCALE)	作本地化识别(locale-aware)匹配
re. M(MULTILINE)	多行匹配，影响^和 $
re. X(VERBOSE)	该标志通过给予更灵活的格式以便将正则表达式写得更易于理解
re. U	根据 Unicode 字符集解析字符，这个标志影响\w,\W,\b,\B

用法如下：

```
>>>import re                              #导入模块
#生成正则对象,匹配单词 python,不区分大小写
>>>rx = re.compile(r'python',re.I)
>>>rx.findall("python Python Jython")     #调用 findall 方法匹配字符串 s
['python', 'Python']                      #返回结果
```

2. re.findall()

遍历匹配，可以获取字符串中所有匹配的字符串，返回一个列表，如果有分组，则返回分组内容。

函数语法为

```
re.findall(pattern, string, flags = 0)
```

前面我们说过，除了编译过的正则对象，这些函数也可以被直接调用，上面的例子也可以是这样：

```
>>>re.findall(r'python', "python Python Jython", re.I)
['python', 'Python']
```

3. re.sub()

使用 re 替换 string 中每一个匹配的子串后返回替换后的字符串。



语法格式为

```
re.sub(pattern, repl, string, count)
```

re. sub 还允许使用函数对匹配项的替换进行复杂处理。下面我们用两种方法分别将字符串中的空格替换为"-"和"[]"。用法如下：

```
>>>import re
>>>text = "I am milo,how are you!"
>>>re.sub(r'\s+', '-', text)
'I-am-milo,how-are-you!'
>>>re.sub(r'\s+', lambda m:'['+m.group(0)+']', text,0)
'I[ ]am[ ]milo,how[ ]are[ ]you!'
```

4. re.match

re. match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功，match()就返回 none。

函数语法为

```
re.match(pattern, string, flags = 0)
```

用法如下：

```
>>>import re
>>>print(re.match('www', 'www.abcdefg.com').span())
(0, 3)
>>>print(re.match('com', 'www.abcdefg.com'))
None
```

5. re.search

re. search 扫描整个字符串并返回第一个成功的匹配。

函数语法为

```
re.search(pattern, string, flags = 0)
```

用法如下：

```
>>>import re
>>>print(re.search('www', 'www.abcdefg.com').span())
(0, 3)
>>>print(re.search('com', 'www.abcdefg.com').span())
(12, 15)
```

6. match object 对象方法

match 和 search 一旦匹配成功，就是一个 match object 对象，而 match object 对象有以

下方法。

(1) group():返回被正则匹配的字符串。

(2) start():返回匹配开始的位置。

(3) end():返回匹配结束的位置。

(4) span():返回一个元组包含匹配(开始,结束)的位置。

(5) group():返回与正则整体匹配的字符串,可以一次输入多个组号,对应组号匹配的字符串。

(6) groupdict():以字典形式返回正则分组匹配的字符串,组名为 key。

15.3 实例:一只小爬虫

正则表达式是爬虫利器,学会正则,不写爬虫真是浪费。不管多少语法讲解都不如实战来得直观,下面我们就来实现一个简单的小爬虫,看一下爬虫基本的工作原理,算是一个入门的敲门砖,要想完成更复杂的爬虫,还要多用些心才行。

网络爬虫是一种按照一定的规则,自动抓取万维网信息的程序或者脚本。爬虫的基本工作有四步。

(1) 明确目标:要知道你准备在哪个范围或者网站去搜索什么样的数据。

(2) 爬:将所有网站的内容全部爬下来。

(3) 过滤:去掉对我们没用处的数据。

(4) 处理数据:按照我们想要的方式存储和使用。

现在就可以先定一些目标,比如我们可以爬取中国天气网未来一周全国省会城市的天气,或者很多人喜欢爬豆瓣的影评等,这里我们把目标定的具体直观一些,比如我们就爬取糗事百科热图首页的段子图片,并且保存到本地。

基本的过程如下。

(1) 先分析网页中图片的地址特征。

(2) 将页面源代码全部抓取过来。

(3) 通过正则表达式匹配出所有图片的地址。

(4) 下载保存到本地。

具体实施的过程如下。

1. 分析

首先分析网页,打开 https://www.qiushibaike.com/imgrank/,可以看到普通的网页,我们需要的是在网页的源代码中找到图片的地址,可以通过在浏览器中按 F12 键看到源代码,如果你用 Chrom,也可以选中一个图片然后右击选择检查,可以直接看到图片的代码部分,如图 15-4 所示。

我们看见图 15-5 中的代码,如果你查看过多个图片的源代码,就会发现相同的要素,整个图片都是包含在<a>(HTML 语言)标签中的,这也是我们获取的关键数据,因为 src=后面第一个引号中的地址就是图片实际的地址。

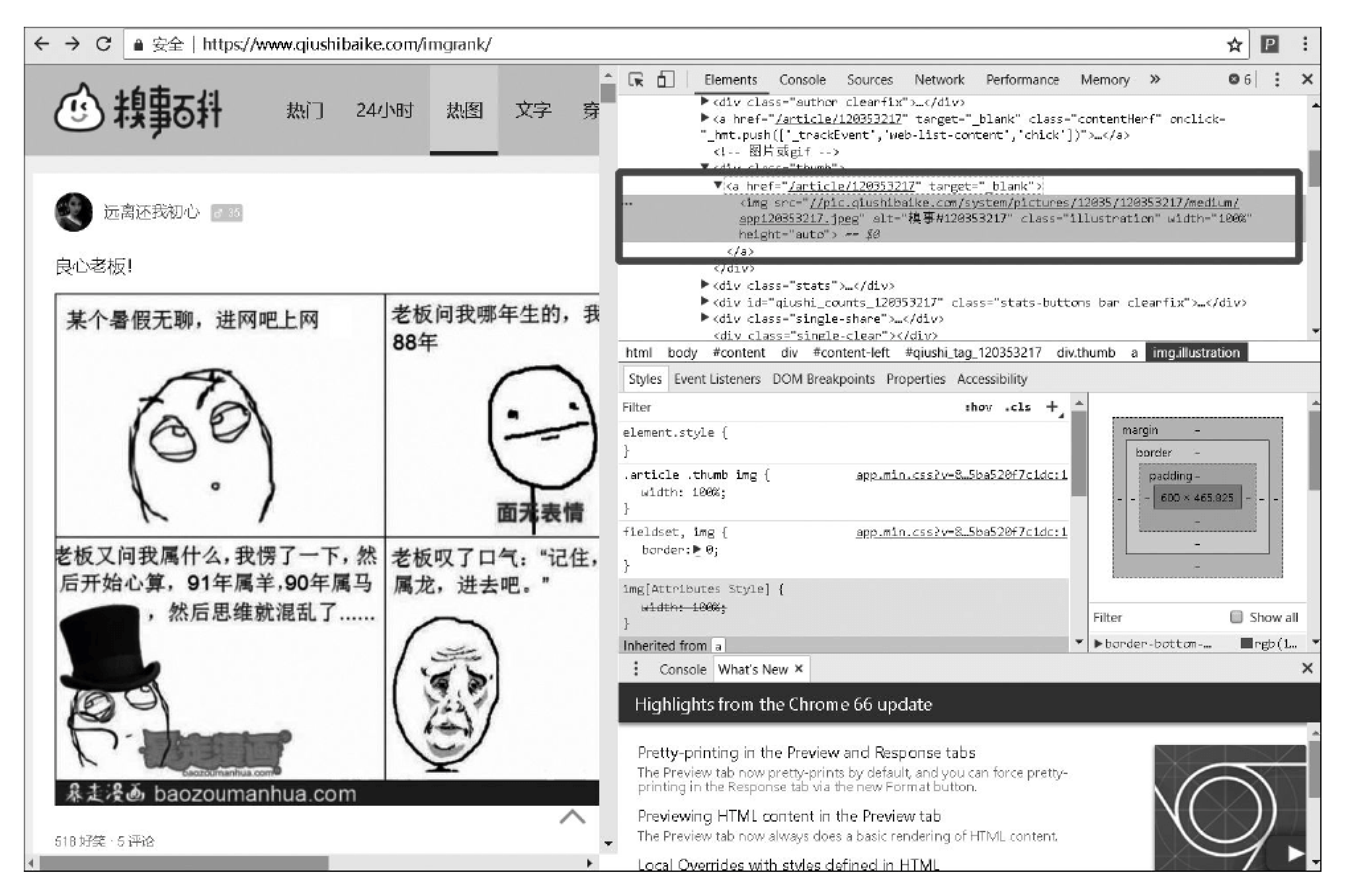


图 15-4　源代码

```
▶<div class="author clearfix">…</div>
▶<a href="/article/120353217" target="_blank" class="contentHerf" onclick=
"_hmt.push(['_trackEvent','web-list-content','chick'])">…</a>
  <!-- 图片或gif -->
▼<div class="thumb">
  ▼<a href="/article/120353217" target="_blank">
     <img src="//pic.qiushibaike.com/system/pictures/12035/120353217/medium/
     app120353217.jpeg" alt="糗事#120353217" class="illustration" width="100%"
     height="auto"> == $0
   </a>
  </div>
▶<div class="stats">…</div>
▶<div id="qiushi_counts_120353217" class="stats-buttons bar clearfix">…</div>
▶<div class="single-share">…</div>
 <div class="single-clear"></div>
```

图 15-5　图片源代码

2. 下载源代码

明白了第一步的目标，接下来就是获取整个网页的源代码了。我们在网络编程的章节介绍过 urllib。所以，按照之前学习的内容可以很容易写出这样的代码获取网页源代码：

```
from urllib import request
url = "http://www.qiushibaike.com/imgrank/"
response = request.urlopen(url)
```

这时你可能会看到下面这样的报错：

```
File
"C:/Users/milo/Desktop/CrazyPythonZeroToHero/code/15/getImgQB.py", line 4, in
<module>
```

```
    response = urllib.request.urlopen(url)
  File
"C:\Users\milo\AppData\Local\Programs\Python\Python36-32\lib\urllib\request.
py", line 223, in urlopen
    return opener.open(url, data, timeout)...
```

这是因为有些网站为了防止被爬虫爬取信息而设置的一个验证，即网站的 UA 防护，一般网站都要检查是否是浏览器在进行访问，所以我们这里的方式就是设置请求头，让网站以为是一个浏览器再访问，下面我们加上请求头再通过 urlopen 打开网页并读取源代码。

```
#getImgQB.py
from urllib import request

url = "http://www.qiushibaike.com/imgrank/"
user_agent = 'Chrome/4.0 (compatible; MSIE 5.5; Windows NT)'
req = request.Request (url,headers = {
    'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
                    })
response = request.urlopen(req)
html = response.read().decode('utf-8')
print(html)
```

这时你会看到 shell 中显示的源代码，但是这没用，接下来才是关键，即我们要针对第一步分析的结果，编写正则表达式。

3. 过滤

获取到原始数据的下一步就是对数据进行过滤，得到有效的信息，我们的目标是源代码中带下划线的部分，也就是图片地址：

```
<a href = "/article/120353217" target = "_blank">
<img src = "//pic.qiushibaike.com/system/pictures/12035/120353217/medium/
app120353217.jpeg" alt = "糗事#120353217" class = "illustration" width = "100%"
height = "auto">
</a>
```

此时编写正则的方式也并不是唯一的，尽可能精确地匹配这个字符串，匹配的规矩要多一些，否则你很可能会匹配到一些没用的图片，比如头像、背景等。

如果用口语描述这个地址，我们可以说图片地址位于“<a 空格+一堆字符+>+换行+<img+huanhang +src=”后面的第一个引号中，同时后面还有一堆字符直到“”。所以最终得出了下面这样一个正则：

```
re.compile('<a.*?>\n<img src = "(.*?)".*?>')
```

可能你已经注意到了“*”和“?”的组合，这涉及正则匹配的模式，试想“<a.”后面的星号，如果没有那个问号，会向后无限匹配，直到最后一个“>”才结束，这样，后面的表达式其实根本就没用，也不会得到想要的结果了，而这里要的仅仅是“<a.”之后一堆字符后的第



一个“>”,所以加了“?”,单独的“?”代表匹配前一字符 0 或 1 次。而跟“ * ”“+”组合使用,则代表非贪婪模式,作最小匹配。这个地方需要好好理解一下,可以在终端多试试,体会一下。

注意:贪婪模式指在整个表达式匹配成功的前提下,尽可能多地匹配(*)。

非贪婪模式指在整个表达式匹配成功的前提下,尽可能少地匹配(?)。

Python 默认是贪婪模式。

另外,我们将图片地址部分的匹配用“()”作了一个分组,这样我们在匹配后只要返回分组数据就可以得到图片地址了,可以先打印到屏幕上。

```
#getImgQB.py
from urllib import request
import re

url = "http://www.qiushibaike.com/imgrank/"
user_agent = 'Chrome/4.0 (compatible; MSIE 5.5; Windows NT)'
req = request.Request(url, headers = {
    'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
                        })
response = request.urlopen(req)
html = response.read().decode('utf-8')
patternImg = re.compile('<a.*?>\n<img src="(.*?)".*?>')
items = re.findall(patternImg, html)

for item in items:
    print(item)
```

结果如下:

```
>>>
//pic.qiushibaike.com/system/pictures/12035/120354571/medium/app120354571.jpeg
//pic.qiushibaike.com/system/pictures/12035/120351986/medium/app120351986.jpg
...
//pic.qiushibaike.com/system/pictures/11692/116921877/medium/app116921877.JPEG
//static.qiushibaike.com/images/web_v3/sidebar/remix_banner.gif?v=
f8bbbe7ca7cd5b9314e8235d6290fb0f
```

我们看到这个结果最后有一条规则的字符串,符合规则的图片后缀名也是分成了 jpeg 与 jpg 两种,还有大、小写的区别,所以最后再对数据作一次过滤,只留下有用的图片地址,并且利用 urllib 直接下载图片文件,文件名用自定义的递增数字表示。

最终的代码封装了两个函数,就像下面这样:

```
#getImgQB.py
from urllib import request
import re

url = "http://www.qiushibaike.com/imgrank/"
```

```
user_agent = 'Chrome/4.0 (compatible; MSIE 5.5; Windows NT)'

def getHtml(url):
    req = request.Request(url, headers = {'User-Agent':user_agent})
    response = request.urlopen(req)
    html = response.read().decode('utf-8')
    #print(html)                                  #测试代码
    response.close()
    return html

def getImg(html):
    patternImg = re.compile('<a.*?>\n<img src="(.*?)".*?>')
    jpglist = re.findall(patternImg, html)
    i = 0
    for item in jpglist:
        patternJpeg = re.compile('.*\.jpe{0,1}g', re.I)
        #print(patternJpeg.findall(item))         #测试代码
        if patternJpeg.findall(item):
            #print("http:"+item)                  #测试代码
            #异常处理作用是防止无效地址导致下载失败
            try:
                request.urlretrieve("http:"+
                                    patternJpeg.findall(item)[0],
                                    "%d.jpg" %i)
            except:
                pass
        i += 1

html = getHtml(url)
getImg(html)
```

在过程中，我们可以用一些 print 语句来随时观察匹配的结果，根据结果作些调整，只要保持思路清晰，基本上是不会有什么问题的。

运行后，就会在你的脚本所在路径看到下载的图片了（见图 15-6）。

图 15-6　爬虫效果

最后需要说明的是，这只是一个基本的爬虫。实际情况可能要复杂，工作量也会比较大。比如，你要爬取整个网站，就需要深入了解爬虫工作原理，要学会使用基本的 http 抓取

工具 scrapy 以及分布式爬虫等一系列的概念和工具。其实技术就是一层纸，现在点透了，想要深入学习就会比较快了。



重点提示

在这一章中，你要掌握的内容：

（1）理解正则表达式的作用。

（2）掌握元字符的意义和用法。

（3）熟悉 re 模块及常用方法。

动动手



解析 apache 日志：正则

（1）仿照本章案例写一个小爬虫来爬取你感兴趣的信息。

（2）在第 6 章的动动手中，apache 日志的空格解析了一次 apache 日志，现在我们用正则的方式再来解析一次，感兴趣的话，可以扫描旁边的二维码观看视频。

（3）通过多线程的方式，直接爬取糗事百科文字的前十页，每页一个线程。

拓展案例篇

第 16 章

小白也玩大数据

对于大数据(Big Data),研究机构 Gartner 给出了这样的定义:大数据是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力来适应海量、高增长率和多样化的信息资产。

大概 2012 年以后,大数据这个词可以说是越来越热门,也被更多的人所知晓。但其实大数据概念由来已久,最早提出"大数据"时代到来的是全球知名咨询公司麦肯锡,麦肯锡称:"数据,已经渗透到当今每一个行业和业务智能领域,成为重要的生产因素。人们对于海量数据的挖掘和运用,预示着新一波生产率增长和消费者盈余浪潮的到来。"

"大数据"在物理学、生物学、环境生态学等领域以及军事、金融、通信等行业存在已久,之所以近年来引起人们关注,还得益于互联网和信息行业的发展。

大数据涉及的技术很多,在这一章,我们会对大数据有一个基本的了解,并且通过 Python 实现一个大数据的小模型。

16.1 好玩的大数据

生活中经常看到大数据的影子,而且通常都会让人觉得很神奇。比如根据你的浏览记录推送你感兴趣的信息,或者在 2017 年年底,很多 APP 都放出了类似个人一年的历程,比如支付宝根据个人消费记录统计出了若干条消费习惯,总结出了你是不是"高富帅"一类的。还有网易云音乐也根据你听歌的时间和歌曲的类型,统计出你是不是有一个夜半听歌的孤独灵魂。

还有很多有趣的大数据案例,我们稍微了解一点。

(1) 一个淘宝数据平台的分析提到,全国购买文胸最多的号码是 B 罩杯,比例占比达 41.45%,其中又以 75B 的销量最好;黑色最畅销。

(2) 沃尔玛经过对用户购物数据分析,男性顾客买了婴儿尿布后,很多会顺手带几瓶啤酒,所以沃尔玛尝试将婴儿尿布和啤酒摆在一个区域,结果尿布和啤酒的销量都大幅增加。如今,"尿布"+"啤酒"的数据分析结果早已成了大数据技术应用的经典案例。

(3) 之前提到过,大数据由来已久,可能你想不到的是,在战争中也有过大数据的成功应用。将军随身笔记本上面记载着每次战斗的缴获、歼敌数量。某次非常小的战役后,将军发现三个问题:

① 缴获的短枪与长枪的比例比其他战役略高;

② 缴获和击毁的小车与大车的比例比其他战役略高；

③ 俘虏和击毙的军官与士兵的比例比其他战役略高。

呃，这有什么用？将军在这次战役的数据启发下，判断出这是对手的指挥所，成功地抓获了敌方将领。

看完这三个小故事，要说的是，大数据技术的战略意义不在于掌握庞大的数据信息，而在于对这些含有意义的数据进行专业化处理。换而言之，如果把大数据比作一种产业，那么这种产业实现盈利的关键在于提高对数据的“加工能力”，通过“加工”实现数据的“增值”。

16.2 大数据技术

了解大数据，首先要搞清楚大数据能做些什么、怎么做。

要介绍大数据就要提到云计算，从技术上看，大数据与云计算的关系就像一枚硬币的正、反面一样密不可分。大数据必然无法用单台计算机进行处理，必须采用分布式架构。它的特色在于对海量数据进行分布式数据挖掘。它必须依托云计算的分布式处理、分布式数据库和云存储、虚拟化技术。也可以说正是因为有了云计算的快速发展，才推动了大数据的应用。

大数据的总体架构包括三层：数据存储、数据处理和数据分析。数据先要通过存储层存储下来，然后根据数据需求和目标建立相应的数据模型和数据分析指标体系对数据进行分析产生价值。

再具体细分大数据的生态圈，涵盖的范围就更广了，其中最出名的就是 Hadoop 了，Hadoop 生态圈或者由其延伸的泛生态系统，基本上都是为了处理大量数据诞生的。

大数据这个圈子里的工具就好像家里的各种家具和物品，各自有各自的功能，可以相互配合，也可能有功能重合的，比如冰箱可以放东西，衣柜也可以放东西，但是你把白菜放在衣柜就发挥不出衣柜的作用；碗可以盛汤，浴缸也可以盛汤，但用浴缸喝汤是不合理的。所以，在大数据领域，并不是标新立异的好地方，即使你强行要创造一些奇异的组合，即使最终完成工作，也不一定是最快、最好的选择。

对传统的单机文件系统来说，横跨不同机器几乎是不可能完成的任务。而通过 HDFS（Hadoop Distributed FileSystem），你可以通过横跨上千甚至上万台机器来完成大量数据的存储，同时这些数据全部都能归属在同一个文件系统之下。

过于庞大的数据，如果只交给一台机器处理，我们可能得等上几周甚至更长时间。以 T 甚至 P 来计量单位的数据，若只靠一台机器，可能等不到结果。

所以使用大量机器进行处理是必然的选择。在大量机器处理过程中，必须处理一些事务：任务分配、紧急情况处理、信息互通等，这时候必须引入 MapReduce/Tez/Spark。其中，前者可以成为计算引擎的第一代产品，后两者则是经过优化后的下一代。MapReduce 采用了非常简单的计算模型设计，可以说只用了两个计算的处理过程，但是这个工具已经足够应付大部分的大数据工作了。

16.3 MapReduce 模型

Hadoop 本身是使用 Java 实现的，所以，最直接的开发就是用 Java 语言。但是对于不懂 Java 语言的开发人员，这又提高了学习成本。好在 Hadoop 提供了 Python 接口，我们可以以更低的学习成本来使用 Hadoop。

Hadoop 中的经典模型 MapReduce，Python 也有这两个函数，而且作用相仿。MapReduce 模型其实就是分成 Map(映射)和 Reduce(规约)两部分合作。比如要统计一个巨大的存储在类似 HDFS 上的文本文件，你想知道这个文本里各个词的出现频率。启动一个 MapReduce 程序后，Map 部分通过几百台机器同时读取这个文件的各个部分，分别把各自读到的部分统计出词频，这几百台机器各自产生一个结果，然后又有几百台机器启动 Reduce 处理，这些 Reduce 将结果汇总，你就得到了整个文件的词频结果。

我们在后面的案例中就通过 Python 手动实现这个 MapReduce 模型，以此作为了解大数据的途径。因为用 Hadoop 框架实现分布式的大数据模型需要先搭建配置 Hadoop 环境，还需要虚拟机，这些已经远远超过“一个章节玩大数据”的范围了，所以我们将模型最小化，只在一台主机上实现通过 MapReduce 模型处理大数据。

16.4 案例：实现 MapReduce 模型

大数据处理的内容没有局限，只要你有数据，就可以拿来做各种各样的分析。最常见的分析可能就是日志文件了，日志文件记录了网站的所有访问行为，通过分析日志能够对网站用户有更进一步的了解从而改善网站的服务。互联网应用中数亿条的访问记录是非常常见的。

如果你没有日志文件，可以找一些其他比较大的文本文件。曾有人用大数据分析过苏轼，下面我们就用大数据的方法分析一下《李太白集》，看看会有什么结果。

16.4.1 案例设计

因为是要模拟 Hadoop 的 MapReduce 模型，虽然没有分布式环境，但我们要把“戏”做足，数据处理的流程一个也不能少，按照 Hadoop 的风格，我们分三步来做这件事。

第一步：将大文件分割成若干份，每份大小要根据实际情况决定(主要是看计算机的配置，比如内存的大小)。

第二步：将第一步分割出来的文件分别传输到多个主机上，由各个主机上的 map 函数进行处理，map 函数主要就是统计分给自己的这部分文件的词频，每个 map 函数都可以同时处理几个文件，处理后再将结果保存为一个文本文件。

第三步：将各个主机的 map 函数统一交给 reduce 处理后，将得到的结果进行汇总。


libai. txt

16.4.2 分割文件

这里我们用到的《李太白集》是事先保存好的 utf-8 编码的 txt 文件

libai.txt。注意：超大文件可能无法直接打开，好在 Python 可以逐行读取文本文件，我们将每 100 行保存为一个文件。如果是其他大数据文件，可以视内存大小适当调整。另外，因为涉及中文，为保证实验顺利，请确保所有文件都是 utf-8 编码。

```
#DataSplit.py
import os
import os.path

def dataSplit(sFile, dFolder):
    line = 100                              #每个文件保存行数
    datatemp = []                           #缓存列表
    n = 1                                   #分割后的文件序号
    if not os.path.isdir(dFolder):          #判断目标目录是否存在
            os.mkdir(dFolder)               #创建目标目录
    with open(sFile, 'r', encoding = 'utf8') as sf:
        dataline = sf.readline()
        while dataline:                     #判断是否从云文件读取到数据
            for row in range(line):         #读取 100 行
                datatemp.append(dataline)   #向缓存添加一行
                dataline = sf.readline()
                if not dataline:            #原文件读取不到数据时结束循环
                    break
            dfilename = os.path.join(dFolder,
                                    os.path.split(sFile)[1] +
                                    str(n) + ".txt")

            with open(dfilename, 'w', encoding = 'utf8') as df:
                df.writelines(datatemp)     #将缓存写入目标文件
                datatemp = []               #清空缓存
                print("%s 创建成功" %dfilename)
                n += 1

if __name__ == "__main__":
    dataSplit("libai.txt", "libai")
```

经过分割后，原本的 1 个文本文件被分割成了 4 个（见图 16-1）。

16.4.3 编写 map 函数

文件分割完毕，就要交给 map 处理了，正常情况下的 map 都是分布处理的，所以可以将数据分析的核心代码放在这里。我们要做的就是读取文件、过滤数据、分析词频，将得到的结果再另存为一个文件。因为这个文件只包含关键词和词频，所以要比原文件小得多，最后再将这些新生成的小文件合成一个文件交给 reduce，reduce 拿到各主机的小文件再汇总合成一个大文件。

这里涉及一个问题，就是我们要读取中文，并且统计中文词语的词频，怎么实现呢？还是要感谢强大的 Python，Python 中有一个专门用来统计中文词频的 jieba 库，可以统计单字也可以统计词语，只需要 pip install jieba 安装上就可以用了。

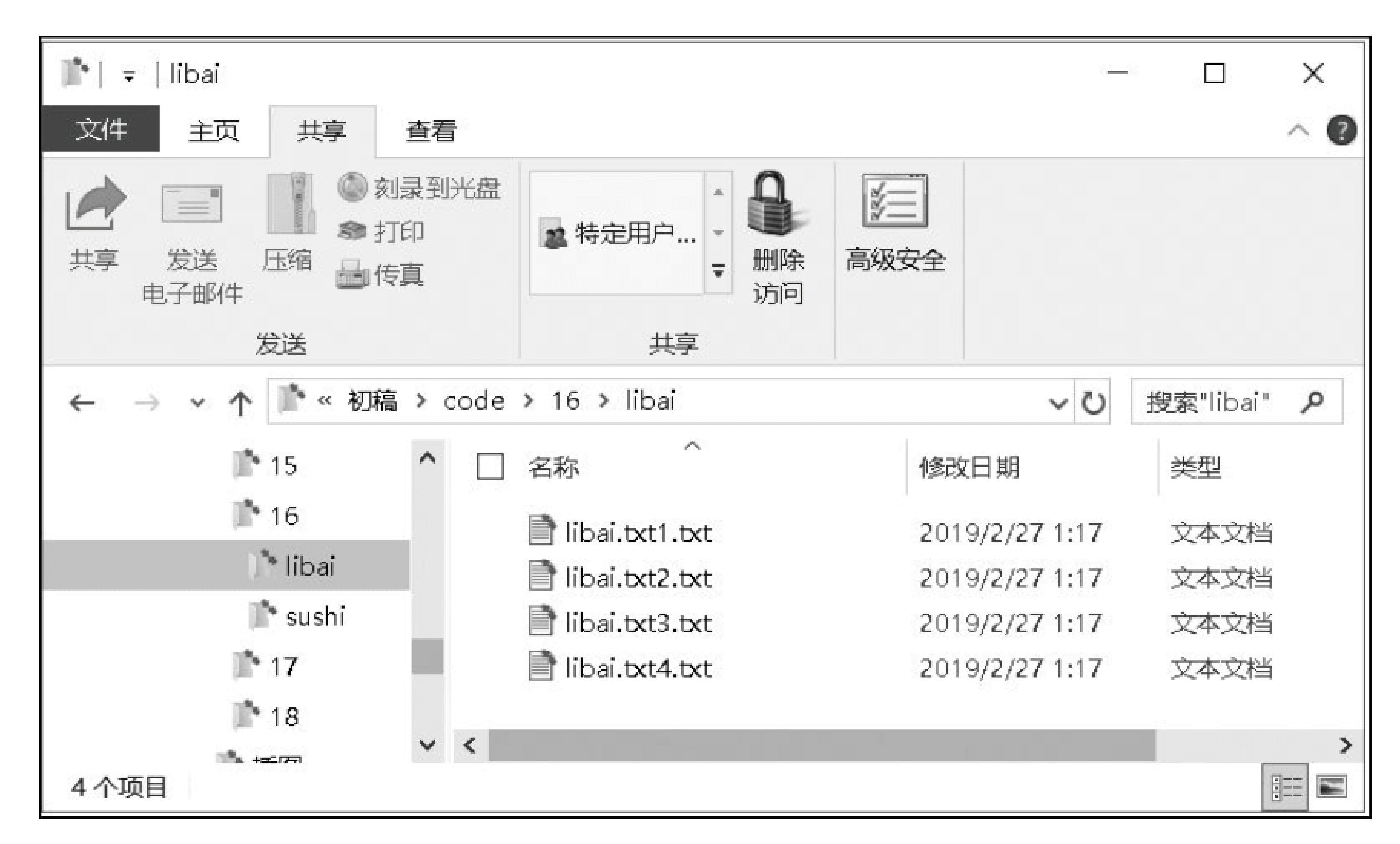


图 16-1 文件分割结果

jieba 有精确模式、全模式、搜索引擎模式三种，其具体算法我们就不研究了，实际用起来很简单，只要把字符串给它，它就会自动判断词语，然后返回列表生成器。为了方便看到效果，我们先用一首《江城子》看一下 jieba 几种模式的分析结果。

```
#jcz.py
import jieba
import jieba.analyse

text = """苏轼《江城子》乙卯正月二十日夜记梦
十年生死两茫茫，不思量，自难忘。千里孤坟，无处话凄凉。
纵使相逢应不识，尘满面，鬓如霜。
夜来幽梦忽还乡，小轩窗，正梳妆。相顾无言，唯有泪千行。
料得年年肠断处，明月夜，短松岗。"""

#seg_list = jieba.cut(text, cut_all = False)   #精确模式(默认模式)
seg_list = jieba.cut(text)
print("[精确模式]: ", "/ ".join(seg_list))

seg_list2 = jieba.cut(text, cut_all = True)    #全模式
print("[全模式]: ", "/ ".join(seg_list2))

seg_list3 = jieba.cut_for_search(text)         #搜索引擎模式
print("[搜索引擎模式]: ","/ ".join(seg_list3))

tags = jieba.analyse.extract_tags(text, topK = 5)
print("[关键词]:", " / ".join(tags))
```

执行结果：

```
[精确模式]:
/ 苏轼/《/ 江城子/ 》/
```

```
/ 乙卯 / 正月 / 二十 / 日夜 / 记梦 /
/ 十年 / 生死 / 两 / 茫茫 / ,/ 不 / 思量 / ,/ 自 / 难忘 / 。/ 千里 / 孤坟 / ,/ 无处 / 话 / 凄凉 / 。/ 纵
使 /相逢 / 应不识 / ,/ 尘 / 满面 / ,/ 鬓 / 如霜 / 。/
/ 夜来 / 幽梦 / 忽 / 还乡 / ,/ 小轩窗 / ,/ 正梳妆 / 。/ 相顾 / 无言 / ,/ 唯有 / 泪千行 / 。/ 料得 / 年 /
年 / 肠断 / 处 / ,/ 明月 / 夜 / ,/ 短 / 松岗 / 。
[全模式]:
/ 苏轼 / / / 江城 / 江城子 / 城子 / /
/ 乙卯 / 正月 / 二十 / 二十日 / 十日 / 日夜 / 记 / 梦 /
/ 十年 / 生死 / 两 / 茫茫 / / / 不 / 思量 / / / 自 / 难忘 / / / 千里 / 孤坟 / / / 无处 / 话 / 凄凉 / / / 纵
使 /相逢 / 应 / 不识 / / / 尘 / 满面 / / / 鬓 / 如 / 霜 / /
/ 夜来 / 幽梦 / 忽 / 还乡 / / / 小轩窗 / / / 正梳妆 / 梳妆 / / / 相顾 / 无言 / / / 唯有 / 泪千行 / 千行 /
/ / 料得 / 年 / 年 / 肠断 / 处 / / / 明月 / 月夜 / / / 短 / 松岗 / /
[搜索引擎模式]:
/ 苏轼 /《/ 江城 / 城子 / 江城子 / 》/
/ 乙卯 / 正月 / 二十 / 日夜 / 记梦 /
/ 十年 / 生死 / 两 / 茫茫 / ,/ 不 / 思量 / ,/ 自 / 难忘 / 。/ 千里 / 孤坟 / ,/ 无处 / 话 / 凄凉 / 。/ 纵
使 /相逢 / 不识 / 应不识 / ,/ 尘 / 满面 / ,/ 鬓 / 如霜 / 。/
/ 夜来 / 幽梦 / 忽 / 还乡 / ,/ 小轩窗 / ,/ 梳妆 / 正梳妆 / 。/ 相顾 / 无言 / ,/ 唯有 / 千行 / 泪千行 / 。
/ 料得 / 年 / 年 / 肠断 / 处 / ,/ 明月 / 夜 / ,/ 短 / 松岗 / 。
[关键词]: 小轩窗 / 正梳妆 / 泪千行 / 松岗 / 江城子
```

从结果来看,精确模式最符合我们的要求,但是需要用正则去掉标点和空白。所以下面选择精确模式,通过字典作计数器。

下面再回来处理我们刚分割的文件《李太白集》。

```
#Map.py
import jieba
import os
import os.path
import re

def Map(sFile, dFolder):
    datatemp = {}
    if not os.path.isdir(dFolder):        #判断目标目录是否存在
        os.mkdir(dFolder)                 #创建目标目录
    with open(sFile, 'r', encoding = 'utf8') as sf:
        datatext = sf.read()
        seg_list = jieba.cut(datatext)    #精确模式(默认是精确模式)
    for w in seg_list:
        if w in datatemp:
            datatemp[w] += 1
        else:
            datatemp[w] = 1

    dList = []
    for k,v in datatemp.items():
        if not re.findall('[\n,、()"":!。《》\s]', k):
            dList.append(k + ' ' + ' ' + str(v) + '\n')
    dfilename = os.path.join(dFolder,
```

```
                                    os.path.split(sFile)[1] + ".map.txt")
        with open(dfilename, 'w', encoding = 'utf8') as df:
            df.writelines(dList)             #将缓存写入目标文件
            print("%s 处理完毕" %dfilename)

    Map("libai\libai.txt1.txt",'libai')
    Map("libai\libai.txt2.txt",'libai')
    Map("libai\libai.txt3.txt",'libai')
    Map("libai\libai.txt4.txt",'libai')
```

这段代码的工作就是读取分割完毕的小文件，提取词语用字典作计数器，再用正则过滤掉标点和空白等无用信息，最后将合格的统计数据另存储到一个文件中。

现在假设这些文本文件经网络聚到 reduce 所在主机（实际这个实验我们就是一台主机），接下来由 reduece 进行数据合并。

```
#Reduce.py
import os
import os.path

def Reduce(sFolder, dFile):
    datatemp = {}
    for root,dirs,files in os.walk(sFolder):
        for f in files:
            if f.endswith(".map.txt"):
                with open(os.path.join(root,f),'r', encoding = 'utf8') as sf:
                    for line in sf:
                        #print(line)
                        k,v = line.split()
                        #print(line)
                        if k in datatemp:
                            datatemp[k] = int(datatemp[k]) + int(v)
                        else:
                            datatemp[k] = v
    dList = []
    for k,v in datatemp.items():
        dList.append(k + ' ' + ' ' + str(v) + '\n')

    with open(dFile, 'w', encoding= 'utf8') as df:
        df.writelines(dList)    #将缓存写入目标文件
        print("%s 合并成功" %dFile)

if __name__ == "__main__":
    Reduce("libai",'rlibai.txt')
```

这里仅仅将每个小文件中的数据读取出来合并数据，最终存储为一个文件 rshijiu.txt，这里面就是我们关心的数据了，你可以打开看看，但是作为程序员，我们应该有更好的办法，再写个显示的工具吧。

```
#Show.py
def show(filename):
    dictList = []
    textdict = {}
    with open(filename, 'r', encoding = 'utf8') as f:
        for line in f:
            dictList.append(line.split())

        print('%-10s%10s' % ('word', 'count'))
        print('- ' * 25)
        for k,v in dictList:
            if int(v) > 100:      #只显示频率大于 100 的词
                print('%-12s %6s' % (k, v))

show("rlibai.txt")
```

这用的是之前介绍过的方法，显示结果如下：

```
word              count
-------------------------
还                  221
来                  161
月                  323
与                  331
为                  189
我                  321
将                  131
送                  112
之                  217
下                  121
不                  234
人                  234
在                  215
上                  197
有                  184
日                  116
欲                  137
如                  118
见                  106
向                  120
谁                  158
若                  124
时                  113
亦                  108
去                  343
中                  104
归                  131
无                  141
道                  105
笑                  116
```

由于 jieba 算法的原因，这里我们看到的都是一个字，不过，古汉语微言大义、字字千钧。可以看到，在李白的诗中除了“我”“去”这样的字，出现最多的就是“月”了，想起“床前明月光”了吗？不难看出诗人将明月作为寄托。不过这个分析还存在很多问题，要想真正了解李白，仅靠这个肯定是不够的。

16.5 彩蛋：词云

可能你觉得上面的效果不那么酷，那我们就来一个看起来酷炫的显示效果——词云（见图 16-2）。看到图 16-2 的效果，心动了吧。其实最简单的词云实现只需要十行代码（别急着看后面的代码，通常代码少就意味着准备工作要做足），一起来看一下吧。



图 16-2　词云示意图

要想做词云需要装三个包：matplotlib、jieba 和 wordcloud。

如果你做了 16.4 节的案例，前两个包就不用安装了，看起来只要再安装一个 wordcloud 就行了，不过，错误经常会出现在这里，现在就来看一下需要注意和可能碰到的问题。

首先，你可以尝试 pip install wordcloud 直接安装，但直接安装多数情况会报错，可以先在 github 下载 wordcloud 的包，解压缩后，在对应目录下用 python setup.py install 安装。如果成功，第一个坑就过去了，没成功就要看报错信息了，有可能是依赖包的问题，那就安装依赖包，主要是有可能会碰到图 16-3 所示的错误。

```
    error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C
++ Build Tools": http://landinghub.visualstudio.com/visual-cpp-build-tools
```

图 16-3　错误示意图

这个原因写得很清楚，是需要 Visual C++ 14.0，你可以去微软的官网下载安装。

还有一个简单的办法，就是到 https://www.lfd.uci.edu/~gohlke/pythonlibs/#wordcloud 安装 whl 版本的包，一定要选择跟自己系统对应的版本（见图 16-4）。

找到文件路径执行下面这个命令就可以了，但是版本不对就会出现红字报错（见图 16-5）。

安装成功后，只需要十行代码就可以画出词云了，不过，这个跟 MapReduce 是没有关系的，不要弄混了。



图 16-4　wordcloud.whl 包

```
C:\Users\milo>pip install wordcloud-1.4.1-cp36-cp36m-win_amd64.whl
wordcloud-1.4.1-cp36-cp36m-win_amd64.whl is not a supported wheel
on this platform.
```

图 16-5　报错

```
#wordcloud19.py
import matplotlib.pyplot as plt|
from wordcloud import WordCloud
import jieba
filetext = open('libai.txt', encoding = 'utf8').read()
wordlist = jieba.cut(filetext, cut_all = True)
wl_space_split = " ".join(wordlist)
#设置词云属性,并生成词云
my_wordcloud = WordCloud(font_path = "C:\WindowsFonts\simhei.ttf").generate(wl_
space_split)
#以下三行绘图
plt.imshow(my_wordcloud)
plt.axis("off")
plt.show()
```

这个代码出来的效果如图 16-6 所示。



图 16-6　《李太白集》高频词词云

这里就能很直观地看到一些内容了，可以看到，李白的诗里出现最多的是“明月”“春风”“相思”等。对不熟悉李白的人，这样似乎更能直观地感受到诗人浪漫主义的情怀，还能知道他曾去过的一些地方等。

需要注意的是，font_path 后面是你计算机字体库中的字体，系统不同，路径和名字都会有区别，本例选择的是黑体。

如果你想画出图 16-2 所示的形状，就需要加背景图片遮蔽，简单来说就是图片什么样，图云就是什么形状。你可以自己试试。

重点提示

在这一章中，你要掌握的内容：

（1）理解 MapReduce 模型。

（2）理解案例模型实现的思路。



动动手

参考本章案例进行实践。

第 17 章

语音识别技术

语音识别技术也被称为自动语音识别(Automatic Speech Recognition,ASR)。这里提及的语音识别技术,其目的是将人类语音中的词汇内容转换为计算机可读的输入,例如按键、二进制编码或者字符序列,而不是根据声音判断说话人是谁。

语音识别技术的应用非常广泛,包括语音拨号、语音导航、室内设备控制、语音文档检索、简单的听写数据录入等。语音识别技术与其他自然语言处理技术如机器翻译及语音合成技术相结合,可以构建出更加复杂的应用,例如语音到语音的同声传译。另外,在你身边现在也有各种各样的语音识别,比如语音控制手机、语音控制输入法、语音控制操作系统,市面上的智能音响也都是语音识别技术的应用。

因为 Python,语音识别可以以一种非常简单的形式开始,但这只是一个体验的方式,因为简单的语音识别可能会受到噪声干扰而影响实验结果。这一章用到的模块是基于 Python 2 的,我们也仅仅是作为一个体验拓展性的介绍,实际的语音识别开发方式很多,也不局限于 Python 2。

17.1 选择语音识别包

国内的百度、阿里云和讯飞等都提供了语音识别平台的 API(Application Programming Interface,应用程序编程接口)。PyPi 上还有很多用于语音识别的软件包,例如:

```
apiai
google-cloud-speech
pocketsphinx
speech
SpeechRecognition
watson-developer-cloud
PyAudio
```

这些软件包功能各异,其中有一些软件包(如 apiai)提供了内置功能,例如识别说话者意图的自然语言处理,这些内置功能超出了基本的语音识别功能。其他的软件包,比如 google-cloud-speech,只关注语音到文本的转换;speech 仅用于 Windows 系统。

在易用性方面,这里有两个最佳选择:speech 和 SpeechRecognition。

语音识别需要音频输入,获取音频的方式有两种:一种是通过音频文件获取;一种是通过麦克风直接获取。

如果你在 Windows 系统做语音识别开发，用 speech 就非常方便，它可以直接利用 Windows 系统的麦克风和语音识别功能。我们在 17.2 节中会介绍 speech 的用法。

而 SpeechRecognition 则更灵活，SpeechRecognition 可以很便捷地使用麦克风和音频文件，可以在几分钟内启动并运行。如果你想要使用 SpeechRecognizer 访问麦克风，就必须安装 PyAudio 包。

SpeechRecognition 库还封装了几个非常流行的语音识别 API，比如 Google Web Speech API，甚至其中还包含了 Google Web Speech API 的默认密钥，这意味着你可以不必注册登录 Google 就能够运行起来。

SpeechRecognition 包的灵活性和易用性使其成为绝佳选择。如果你想深入研究语音识别技术，我强烈推荐(但是不保证支持它包装的每个 API 的每个功能)。你需要花一些时间深入研究，以确定 SpeechRecognition 是否适用于你的特定情况。

为了让你的第一次语音识别体验更顺畅，下面直接利用 speech 做一些尝试。

17.2 speech 模块

Windows 已经有很成熟的语音识别技术，Python 中有个 speech 模块，提供了一个简单的接口对应 Windows 语音识别技术。我们可以利用 Windows 的语音识别技术，实现将语音转换为文本或者将文本转换为语音。

speech 模块有自己的官方站点：http://python-speech-features.readthedocs.io，有兴趣的可以多了解一下。主要功能包括识别语音并转换成文本；文本合成语音，特定词的识别，比如对获取的声音进行特定词的捕捉；特定发声人特定词的捕捉。下面将实现语音和文字之间的转换。

17.2.1 语音识别开发环境搭建

要在 Windows 中进行语音识别开发，主要需要三个工具 Python、speech 模块以及 Microsoft Speech SDK。

相关软件安装方法如下。

(1) Python 2 或 Python 3：speech 支持 Python 2，所以如果你要用 speech 开发，现在最好使用 Python 2.7。如果你是 Python 3，想体验一下，在后面也会介绍怎么用 Python 3 体验语音识别技术。

(2) Microsoft Speech：如果你的系统版本高于 Windows 8，那么恭喜你，系统是自带语音识别技术的。如果系统版本低，可以到微软的官网下载(https://www.microsoft.com/en-us/download/details.aspx? id=10121)，默认安装就可以了。

(3) pywin32：pywin32 即 Python for Windows Extensions，提供了 Pyhton 访问和调用 Windows 底层功能函数的接口，直接 pip install pywin32 安装就可以了。

(4) speech：直接 pip install speech 就可以，如果失败，可以自行下载 speech.py，并放到 Python 安装路径下的 site-packages 目录就可以了，比如我的就是在这里 C:\\Python\Python36-32\lib\site-packages\speech.py。

17.2.2 环境配置和调试

现在你可以导入 speech 模块来看一下模块是否安装成功(见图 17-1)。

```
>>> import speech
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\milo\AppData\Local\Programs\Python\Python36-32\lib\site
-packages\speech.py", line 157
    print prompt
               ^
SyntaxError: Missing parentheses in call to 'print'
```

图 17-1　导入 speech 模块

如果是 Python 2.7,这里不会有问题;如果是 Python 3,就可能会看到这个报错,原因是安装的 speech.py 是按照 Python 2 来写的,Python 3 中已经不用 print 命令了。若为 Python 3,解决办法就是按报错提示的路径,找到 speech.py 文件:

```
line157(157行), 修改 print prompt,改成 print(prompt)
```

另外,在 Python 3 环境下,还会碰到一个问题,需要进行一点修改(Python 2.7 不需要),修改如下:

```
line59(159行),将 import thread,改成 import threading
class T(threading.Thread):
"""增加这个自定义类"""
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
            pass

def _ensure_event_thread():
    """
    Make sure the eventthread is running, which checks the handlerqueue
    for new eventhandlers to create, and runs the message pump.
    """
    global _eventthread
    if not _eventthread:
        def loop():
            while _eventthread:
                pythoncom.PumpWaitingMessages()
                if _handlerqueue:
                    (context,listener,callback) = _handlerqueue.pop()
                    #Just creating a _ListenerCallback object makes events
                    #fire till listener loses reference to its grammar object
                    _ListenerCallback(context, listener, callback)
                time.sleep(.5)
        _eventthread = T()          #此处修改
        _eventthread.start()        #此处修改
```

修改后,不报错就可以成功导入 speech 了,如果之前没有开启语音支持,如 Windows

10 会自动弹出启动语音识别设置，如果没弹出，也要手动设置一下并且启动，方法如下。

打开控制面板找到语音识别（见图 17-2），启动语音识别就可以了，基本上按默认设置就可以了。



图 17-2　启动语音识别

除了启动语音识别，还有设置麦克风，甚至学习语音教程，另外，在左侧的高级语音设置里面还有启动试运行语音识别、启用语音激活等设置。

启动了系统的语音识别，接下来就是见证奇迹的时刻，确认打开音箱和麦克风可用，你只需要做下面的事情：

```
>>>import speech
>>>speech.say("hello!邹琪鲜!")
>>>speech.say("欢迎使用 speech!")
```

听到计算机里面的女孩跟你打招呼了吗？

17.2.3　文字和声音相互转化

现在只需要调用一个 say 方法就能说话了，这就是 speech 将文字转为语音的方法，你完全可以写一个会讲故事的程序啊。

如果想要识别语音并转化为文字，也是很简单的，只需要一个 speech.input()方法。speech.input()的用法和 input()类似，只不过一个是等待用户输入声音，一个是等待用户输入字符。

```
import speech
saysomething = speech.input("hello! Say something!")
speech.say(saysomething)
```

```
print(saysomthing)
```

这段代码运行后，首先屏幕上显示 hello! Say something!，然后我们对麦克风说一句“你好”，这时语音就会转为文字赋值给 saysomething，你可以让计算机说出来，也可以显示到屏幕上。不过这个实验最好在 Python 2.7 上做，在 Python 3 可能会有些问题。

现在我们已经可以让计算机说话，也可以让计算机听懂我们的话了。你会发现，这就是将输入/输出从字符串变成了语音而已，剩下其他的关于逻辑开发，跟之前没什么两样。

需要说明一点，因为噪声等原因，这个实验的成功率比较低。不过，不必担心没有掌握这项技术，前面我们说过，这是一个体验性的实验，真正要做语音开发还有很多方式可以选择，比如 SpeechRecognition。

17.2.4 speech 模块的其他方法

在这里仅仅是通过一个小工具看到了 Python 的便捷之处，且不说其他开发接口，单 speech 就还有很多实用的方法，比如下面这些。

speech.listenforanything(callback)：听到任何信息就启动一个独立线程 callback。

speech.listenfor(phraselist,callback)：将指定语音转化为文字。

speech.listener.islistening()：指定特定听者处于监听状态。

speech.listener.stoplistening()：指定特定听者处于关闭状态。

speech.islistening()：开启所有听者。

speech.stoplistening()：关闭所有听者。

通过 speech 除了让计算机讲故事，还可以利用 speech 语音控制计算机。如果你对语音技术感兴趣，那就大胆探索吧。

重点提示

在这一章中，你要掌握的内容：

在这一章我们通过一个容易上手的模块来了解语音识别技术的应用，可以参考本章案例进行练习。若想要深入了解，可以在线了解语音识别 API，比如百度就提供了 Python 接口以及详细的使用说明。



动动手

尝试一下看起来很神奇的语音识别吧！

第 18 章 六行代码入门机器学习

人工智能(AI)越来越被大家关注,即便不是专业的从业人员,也都会从各种新闻中看到人工智能的影子,比如无人驾驶、人工智能跟人类棋手下棋等。

机器学习(Machine Learning)作为人工智能的一个分支,名字可能没有人工智能响亮,但其相关技术已经在我们生活中的很多地方得以应用,比如:

(1) 网站或淘宝根据浏览过的记录推送相似内容、广告。

(2) 医院的 CT 通过人工智能检测阴影。

(3) 手机、智能电视的语音控制、语音转化为文字、人脸识别。

(4) 一封电子邮件是否是垃圾邮件。

(5) 一篇文章应该分到科技、政治,还是体育类,一段文字表达的是积极的情绪还是消极的情绪。

(6) 测量市场营销的成功度。

(7) 预测某个产品的收益,股票。

(8) Google 预测到 H1N1 爆发,百度预测 2014 世界杯。

(9) 人类基因剪接位点识别、基于图片的性别检测、大规模图片分类。

这一章内容会从人工智能的一个简单案例作为突破口,希望能够为你学习人工智能打开一扇门,有一个轻松的开始。

18.1 人工智能发展简史

在谈论机器学习之前,需要先搞清楚几个概念,因为一说到机器学习,就会有人摆出来一堆概念,搞得新手无从下手,什么人工智能、机器学习、数据挖掘、深度学习等。那它们到底是什么关系呢?

人工智能是最早出现的,也是最大的集合(包含其他概念)。1956 年,几位计算机科学家相聚在达特茅斯会议(Dartmouth Conferences),提出了“人工智能”的概念。人工智能曾一度被极为看好。之后的几十年,人工智能一直在两极反转,或被称作人类文明耀眼的未来;或被当成技术疯子的狂想扔到垃圾堆里。过去几年,尤其是 2015 年以来,人工智能开始大爆发。很大一部分是由于 GPU 的广泛应用,使并行计算变得更快、更便宜、更有效。当然,无限拓展的存储能力和骤然爆发的数据洪流(大数据)的组合拳也使图像数据、文本数据、交易数据、映射数据全面海量爆发。

人工智能最开始要实现的是“强人工智能”(General AI),也就是通过计算机构造复杂的、拥有与人类智慧同样本质特性的机器。通常科幻电影里有了自我意识要消灭人类的就是这种,但实际上,现阶段来看这是不可能的。我们目前能实现的,一般被称为“弱人工智能”(Narrow AI)。弱人工智能是能够与人一样,甚至比人更好地执行特定任务的技术。例如,Pinterest 上的图像分类或者 Facebook 的人脸识别。这些技术实现的是人类智能的一些具体的局部。但它们是如何实现的?这种智能是从何而来?这就是机器学习。

机器学习是一种实现人工智能的方法,其最基本的做法是使用算法解析数据、从中学习,然后对真实世界中的事件做出决策和预测。与传统的为解决特定任务、硬编码的软件程序不同,机器学习是用大量的数据来“训练”,通过各种算法从数据中学习如何完成任务。

机器学习直接来源于早期的人工智能领域。传统算法包括决策树学习、推导逻辑规划、聚类、强化学习和贝叶斯网络等。众所周知,我们还没有实现强人工智能。早期机器学习方法甚至都无法实现弱人工智能。

机器学习目前最成功的应用领域应该就是计算机视觉了,比如识别车牌号。

深度学习(Deep Learning)是一种实现机器学习的技术,是由早期机器学习中的一个重要算法——人工神经网络(Artificial Neural Networks)发展出来的。神经网络的原理是受我们大脑的生理结构(互相交叉相连的神经元)启发。但与大脑中一个神经元可以连接一定距离内的任意神经元不同,人工神经网络具有离散的层、连接和数据传播的方向。

其实在人工智能出现的早期,神经网络就已经存在了,但神经网络对于“智能”的贡献微乎其微。主要问题是,即使是最基本的神经网络,也需要大量的运算,神经网络算法的运算需求难以得到满足。直到 GPU 得到广泛应用,深度学习才得以快速发展。比如 2012 年吴恩达(Andrew Ng)教授在 Google 实现了神经网络学习到猫的样子。吴教授的突破在于,把这些神经网络从基础上显著地增大了。层数非常多,神经元也非常多,然后给系统输入海量的数据来训练网络。在吴教授这里,数据是一千万 YouTube 视频中的图像。吴教授为深度学习加入了“深度”(deep)。这里的“深度”就是说神经网络中众多的层。

现在,经过深度学习训练的图像识别,在一些场景中甚至可以比人做得更好:从识别猫,到辨别血液中癌症的早期成分,到识别核磁共振成像中的肿瘤。Google 的 AlphaGo 先是学会了如何下围棋,然后与自己下棋训练。它训练自己神经网络的方法就是不断地与自己下棋,反复地下,永不停歇。

随着计算机计算力的增加,人工智能领域得到更加飞速的发展,或许很快就会有“终结者”出现了。

18.2 机器学习初体验:搭建机器学习环境

用一句话概括机器学习就是机器(计算机)通过获取新知识和新技能,识别现有知识。

通过对人工智能发展的了解,相信你已经发现,机器学习是目前最能产生实际作用的一个分支,而且可以通过机器学习再向深度学习递进。而学习最快的途径就是直接动手,所以我们现在就开始吧。

机器学习比较流行的框架有两个:scikits-learn 和 TensorFlow。scikits-learn 主要适

合中小型、实用机器学习项目，尤其是数据量不大且需要使用者手动对数据处理，并选择合适模型的项目，这类项目使用 CPU 即可。TensorFlow 适合已经明确需要用深度学习，且数据处理需求高的项目，这类项目往往数据量较大，且最终需要的精度高，需 GPU 加速运算。

下面通过 scikits-learn 了解机器学习。

scikits-learn 有自己的官方网站(见图 18-1)，可以在上面找到环境搭建的方法。

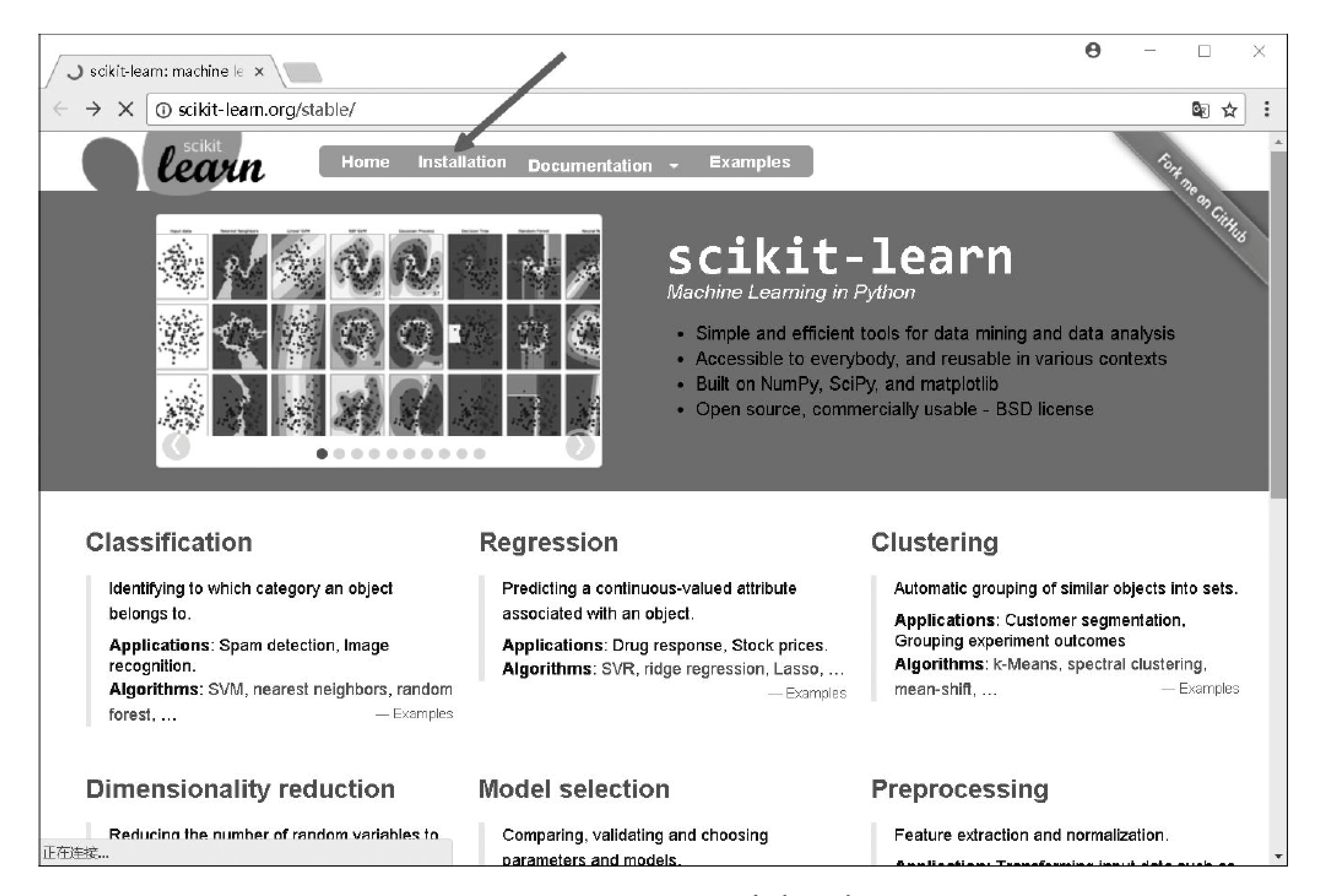


图 18-1　scikits-learn 官方网站

你会找到下面的安装说明：

```
Installing the Scikit- learn
Scikit- learn requires:
    Python (>= 2.7 or >= 3.3),
    NumPy (>= 1.8.2),
    SciPy (>= 0.13.3).
If you already have a working installation of numpy and scipy, the easiest way to
install scikit-learn is using pip
    pip install - U scikit-learn
```

怎么样？没那么复杂吧，而且，官方说明还提到了一个办法，就是安装 Anconda 集成环境。Anaconda 指一个开源的 Python 发行版本，包含了 conda、Python 等 180 多个科学包及其依赖项。因为包含了大量的科学包，Anaconda 的下载文件比较大(约 515MB)，如果只需要某些包，或者需要节省带宽或存储空间，也可以使用 Miniconda 这个较小的发行版(仅包含 conda 和 Python)。可以到 Anaconda 官方网站下载，注意 Python 版本。

安装好之后就可以使用了，其实就是一个模块，import 不报错就可以开始下一步了，这

也是我们的第一行机器学习代码。

```
>>>import sklearn
```

18.3 机器学习的过程

先考虑一个问题,你怎样判断你面前的物体是苹果还是橘子(见图 18-2)? 机器又是怎么怎么判断的呢?




图 18-2 苹果和橘子

前面我们说了,机器学习就是使用算法来解析数据、从中学习,然后对真实世界中的事件做出决策和预测。所以,机器学习可以分成三步实现。

(1) 收集训练数据(Collect Training Data)。

(2) 训练分类器(Train Classifier)。

(3) 做出预测(Make Predictions)。

让机器判断物体是苹果还是橘子就是我们面临的问题,为了解决这个问题,需要写一个功能来区分水果。

(1) 将描述水果的属性作为输入数据。

(2) 根据重量、质感等特性对收集的数据进行预测这个水果是苹果还是橘子。

18.3.1 收集训练数据

为了收集数据,我们想象超市的水果货架,看苹果和橘子的不同之处,想想你是怎么分辨苹果和橘子的,然后把收集好的数据记录到一张表中。机器学习中这种测量得到的值叫作特征(Features),为简单起见,我们采取两个数据:重量和质感,选择合适的特征才能更好地区分水果种类,如表 18-1 所示。

表 18-1 采集数据表

重量	质感	标签
150g	Bumpy	Orange
170g	Bumpy	Orange
130g	Smooth	Apple
140g	Smooth	Apple
…	…	…

表格的每一行都是我们训练数据的一个样本，最后一列称为标签，用来表明这个样本是什么水果，整张表格就是我们的训练数据。

训练数据越多，分类器就越能更好地工作，我们拥有越多的训练数据，预测的结果也就越准确。现在先把训练数据转化为代码。

定义两个变量：特征和标签。

```
features = [[150, 'bumpy'],[170, 'bumpy'],
            [130, 'smooth'],[140, 'smooth']]
labels = ['orange', 'orange', 'apple', 'apple']
```

为了让数据简洁，我们自定义规则转化一下数据，smooth 和 orange 用 1 表示，bunpy 和 smooth 用 0 表示。这样就有了前三行代码。

```
#sk_AppleOrange.py
import sklearn
features = [[150, 0],[170, 0],[130, 1],[140, 1]]
labels = [1, 1, 0, 0]
```

18.3.2　训练分类器并做出预测

收集到数据后，接下来要做的就是使用这些数据训练分类器。分类器有很多类型，或者说有很多种算法，传统算法包括决策树学习、推导逻辑规划、聚类、强化学习和贝叶斯网络等。学习机器学习，本质上是学习算法。

按照训练的数据有无标签，可以将上面提到的算法分为监督学习算法和无监督学习算法。

（1）监督学习算法：决策树、线性回归、逻辑回归、神经网络、SVM。

（2）无监督学习算法：聚类算法、降维算法。

还有一个推荐算法较为特殊，既不属于监督学习，也不属于非监督学习，是单独的一类。这里不对算法做过多介绍，我们直接使用决策树。简单地说，决策树有点像 if，通过一系列的判断得到最终结果。

比如我们将所有数据传给决策树后，决策树就会根据数据进行学习，并得到自己的判断标准，这时我们再给它一个新的数据，它就会给出预测结果。比如给它的数据中大于等于 150 克且粗糙的是橘子，当你给它一个大于等于 150 克且粗糙的数据，它可能就会预测为橘子。说可能，是因为训练数据比较少，不一定精准，比如给它一个 3000 克粗糙的数据，它可能会告诉你这是苹果。

最终代码如下：

```
#sk_AppleOrange.py
from sklearn import tree
"""
Orange->1
Apple->0
features = [[150, 'bumpy'], [170, 'bumpy'],
            [130, 'smooth'],[140, 'smooth']]
```

```
"""
features = [[150, 0],[170, 0],[130, 1],[140, 1]]
labels = [1, 1, 0, 0]
clf = tree.DecisionTreeClassifier()      #生成决策树分类器
clf = clf.fit(features, labels)          #训练分类器
print(clf.predict([[160,0]]))
```

这段代码实际只有 6 行：

(1) 从 sklearn 中导入了决策树 tree。

(2) 定义了特征。

(3) 定义了标签。

(4) 生成决策树分类器。

(5) 输入数据训练分类器。

(6) 提供一个新的数据让分类器作预测。

最终的结果，请你动手试一下吧。

通过这个简单的过程，希望能够让你对机器学习有一个初步了解，如果想深入学习机器学习，也不会那么恐惧，Python 提供了很好的工具，你要做的就是一点一点地在实践中学习了。

重点提示

在这一章中，你要掌握的内容：

本章所介绍的内容可以作为机器学习的启蒙内容，希望能够让你对机器学习有一个跨过门槛的认识。如果想要深入研究机器学习，就不是本书的目的了，可以再参考其他专业机器学习的书籍。



动动手

尝试本章的代码，初步体会一下机器学习吧。

```
"""
features = [[150, 0],[170, 0],[130, 1],[140, 1]]
labels = [1, 1, 0, 0]
clf = tree.DecisionTreeClassifier()      #生成决策树分类器
clf = clf.fit(features, labels)          #训练分类器
print(clf.predict([[160,0]]))
```

这段代码实际只有 6 行：

（1）从 sklearn 中导入了决策树 tree。

（2）定义了特征。

（3）定义了标签。

（4）生成决策树分类器。

（5）输入数据训练分类器。

（6）提供一个新的数据让分类器作预测。

最终的结果，请你动手试一下吧。

通过这个简单的过程，希望能够让你对机器学习有一个初步了解，如果想深入学习机器学习，也不会那么恐惧，Python 提供了很好的工具，你要做的就是一点一点地在实践中学习了。

重点提示

在这一章中，你要掌握的内容：

本章所介绍的内容可以作为机器学习的启蒙内容，希望能够让你对机器学习有一个跨过门槛的认识。如果想要深入研究机器学习，就不是本书的目的了，可以再参考其他专业机器学习的书籍。



动动手

尝试本章的代码，初步体会一下机器学习吧。

参 考 文 献

[1] Chun W J. Python 核心编程[M]. 2 版. 张波翔，李斌，李晗，译. 北京：人民邮电出版社，2008.
[2] William F Punch，Richard Enbody. Python 入门经典：以解决计算问题为导向的 Python 编程实践[M]. 张敏，译. 北京：机械工业出版社，2012.
[3] Allen B Downey. 像计算机科学家一样思考 Python[M]. 赵普明，译. 北京：人民邮电出版社，2013.